VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Znovu použitelnost výsledků vyhledávání v rozsáhlých databázích

Reusability of Search Results in Large Databases

Bc. Pavla Polová

Diplomová práce

Vedoucí práce: Ing. Radoslav Fasuga, Ph.D.

Ostrava, 2021

Abstrakt

Předmětem této diplomové práce je prozkoumání metod znovu použitelnosti výsledků provedených vyhledávání nad rozsáhlou databází, a to zejména za použití cache v operativní paměti. Nejprve jsou analyzovány databázové systémy MySQL, MongoDB a Elasticsearch, které budou v práci použity. Následně je stručně vysvětlen princip cache a jsou popsány základní mechanismy, jako je plnění cache, uvolňování cache a problematika aktualizace dat. Dále jsou uvedeny a porovnány různé technologie, které je možné za účelem cachování použít.

V praktické části je zdokumentováno provedené testování tří databázových systémů, na kterých byly spouštěny unikátní nebo opakující se dotazy. Bylo měřeno, jak se mění rychlost zpracování těchto dotazů s použitím cachování a bez něj. Výstupem tohoto testování bylo shrnutí výsledků a doporučení, který databázový systém se hodí pro které použití, a zhodnocení, jaký přínos mělo cachování výsledků databázového hledání a jestli se vyplatí tento mechanismus implementovat.

Klíčová slova

databáze, SQL, NoSQL, vyhledávání, MySQL, Elasticsearch, MongoDB, cache, Redis

Abstract

The subject of this master thesis is to explore possible solutions of reusability of search results in a large database, especially by using cache in computer memory. Firstly, the database systems MySQL, MongoDB and Elasticsearch that will be used in this thesis were described. Then the principle of cache was briefly explained and the basic mechanisms, such as cache admission, data eviction and the data invalidation problem, were described. Next, various technologies for caching were introduced.

In the practical part, testing of three database systems was documented, executing either unique or repeating queries on each of these systems. It was measured how the execution time changes with or without the use of cache. The outcome of this testing was the result summary and recommendation of which database system is best for which case, as well as the evaluation of the benefits of caching the database query results and if it is worth it to implement this mechanism.

Keywords

database, SQL, NoSQL, search, MySQL, Elasticsearch, MongoDB, cache, Redis

Poděkování

Ráda bych poděkovala Ing. Radoslavu Fasugovi, Ph.D. za odborné vedení, věcné připomínky a vstřícnost při konzultacích.

Obsah

Seznam použitých zkratek a symbolů

ACID	–	Atomicity, Consistency, Isolation, Durability
ASCII	–	American Standard Code for Information Interchange
BSON	–	Binary JSON
CLI	–	Command Line Interface
CRUD	–	Create, Read, Update, Delete
CSV	–	Comma-separated values
DB	–	Database
FIFO	–	First In, First Out
GPL	–	General Public License
HTML	–	HyperText Markup Language
ID	–	Identifikátor
JSON	–	JavaScript Object Notation
JVM	–	Java Virtual Machine
LFU	–	Least Frequently Used
LRU	–	Least Recently Used
MRU	–	Most Recently Used
NoSQL	–	Non-SQL, non-relational
OCPU	–	Oracle Compute Unit
RAM	–	Random Access Memory
ROWID	–	Row ID - identifikátor záznamu, též RID
RR	–	Random Replacement
SHA	–	Secure Hash Algorithm
SQL	–	Structured Query Language
SŘBD	–	Systém Řízení Báze Dat
SSD	–	Solid-state drive
TTL	–	Time To Live
vCPU	–	Virtual CPU
WAN	–	Wide Area Network

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

Kapitola 1

Úvod

Tato práce se zabývá způsoby opakovaného využívání výsledků vyhledávacích dotazů v rozsáhlých databázích. Databáze jsou často nejpomalejším článkem celé infrastruktury informačního systému, zejména pokud obsahují velké množství dat. Kromě optimalizací samotné databáze a zvyšování výpočetní kapacity hardwaru lze dosáhnout lepšího výkonu systému také omezováním množství dotazů, které musí databáze zpracovat. Základní myšlenkou je použití paměti cache na serverové straně aplikace, do které se ukládají výsledky již proběhlých hledání.

V této práci jsou nejprve analyzovány tři populární databázové systémy: MySQL jakožto zástupce relačních databází, MongoDB z rodiny NoSQL databází a Elasticsearch jako fulltextový vyhledávač. Součástí je popis jejich základních vlastností i orientační srovnání jejich cen. Následuje popis principu fungování cache a souvisejících mechanismů, jako je plnění a uvolňování cache. Samostatná kapitola je pak věnovaná problému aktualizace dat a s ním souvisejícímu zastarávání dat v cache. V této kapitole jsou nastíněny metody, které je možné použít pro znehodnocení zastaralých výsledků vyhledávání v cache, a popsány výhody i nevýhody, které každý z těchto postupů přináší. Dále jsou uvedeny a srovnány některé technologie, které se využívají pro implementaci cachování. Zvláštní důraz byl kladen na technologii Redis, která je v současné době nejpopulárnějším úložištěm typu klíč-hodnota.

Poslední částí této práce je samotná implementace a testování. Základem pro tuto praktickou část práce je sada dat z existující databáze (Gloffer). Jedná se o produktový katalog, který obsahuje téměř 60 milionů záznamů. Databáze Gloffer je uložená v systému řízení báze dat MySQL, a to jak v relační podobě, tak v podobě JSON dokumentů. Tato data byla importována do databázových systémů, které byly analyzovány v teoretické části práce. Dále bylo pro každou databázi připraveno několik sad testovacích dotazů, které obsahují parametrické a fulltextové vyhledávání i jejich kombinaci. Tyto sady byly také připraveny jak v podobě unikátních dotazů, tak v podobě dotazů, které se s určitou frekvencí opakovaly. Cílem první fáze testování bylo ověřit, zda při spouštění opakujících se dotazů dojde ke zrychlení celkového času provádění ve srovnání s unikátními dotazy díky vnitřních cache mechanismů daného SŘBD. Ve druhé fázi pak bylo implementováno cachování na straně serveru

za použití paměťového úložiště Redis a bylo opět provedeno měření s cílem porovnat, o kolik se v tomto případě sníží celkový čas.

Na závěr jsou porovnány časy celkového trvání provádění daných dotazů bez cachování i s cachováním, a to i napříč jednotlivými databázemi. Výstupem této práce je analýza těchto výsledků a doporučení, které databázové systémy jsou vhodné pro které typy vyhledávání, jakým způsobem ovlivnilo cachování v technologii Redis výkon a jestli je výhodné cachování výsledků databázových hledání implementovat.

Kapitola 2

Přehled použitých SŘBD

Cílem této kapitoly je shrnout důležité informace o technologiích, které byly testovány v rámci této práce. Byli zvoleni 3 zástupci moderních SŘBD, a to MySQL jako zástupce relačních databází, dále MongoDB jakožto dokumentově orientovaná databáze, a nakonec vyhledávací engine Elasticsearch. Pro porovnání byla také u každé technologie vypočtena cena modelového příkladu. Tyto ceny jsou však pouze orientační, neboť ceny u jednotlivých technologií se výrazně liší v závislosti na podrobných požadovaných parametrech a pro konkrétní cenovou nabídku je potřeba kontaktovat příslušnou společnost.

2.1 MySQL

MySQL je multiplatformní SŘBD aktuálně vlastněný společností Oracle Corporation. Na základě srovnání ze serveru DB-Engines se jedná o jeden z nejvyužívanějších a nejlépe hodnocených databázových systémů [1]. MySQL využívá relačního datového modelu a jazyka SQL.

Název licence	**Cena**
MySQL Standard Edition	$2000/rok
MySQL Enterprise Edition	$5000/rok
MySQL Cluster CGE	$10000/rok

Tabulka 2.1: Ceny jednotlivých licencí MySQL Enterprise Server ke dni 10.2.2021 (dostupné na https://www.mysql.com/products/)

Samotný MySQL existuje ve dvou hlavních vydáních. Prvním je MySQL Community Server, který je zdarma dostupný pod GPL licencí. Druhým je MySQL Enterprise Server, který obsahuje stejné jádro (MySQL server), ale doplňuje dodatečné nástroje a podporu 24/7. Druhy licencí MySQL Enterprise Server a jejich ceny jdou uvedeny v tabulce 2.1. S vyšší úrovní licence se přidávají

dodatečné funkce, jako je bezpečnost, monitorování, zálohování, zvýšená zaručená dostupnost nebo škálovatelnost.

Pro využití cloudového úložiště je možné si zakoupit službu Oracle MySQL Database Service. Tabulka 2.2 ukazuje hodinovou a měsíční sazbu pro modelový příklad: s jedinou instancí, úložištěm 240 GB, 2 OCPU a 32 GB paměti.

Název služby	**Cena za hodinu**	**Cena za měsíc**
Oracle MySQL Database Service	$0,1778	$128

Tabulka 2.2: Cena služby Oracle MySQL Database Service pro modelový příklad ke dni 10.2.2021 (vypočteno z https://www.oracle.com/cloud/cost-estimator.html)

MySQL využívá jakožto svůj hlavní úložišťový engine InnoDB. Je to implicitní engine používaný při příkazu `CREATE TABLE` a Oracle doporučuje jeho použití ve většině případech. InnoDB dodržuje transakční pravidla ACID, zahrnuje zamykání na úrovni databázových řádků pro řízení souběhu a vylepšení výkonu a uspořádává data na disku pro optimalizaci minimalizací počtu I/O operací při vyhledávání podle primárního klíče [2].

Pro specifické případy však MySQL nabízí také další enginy, jejichž vlastnosti se od InnoDB výrazně liší. Další významný engine je MyISAM, který na rozdíl od InnoDB nepodporuje transakce a umožňuje pouze zamykání na úrovni celé databázové tabulky. Díky tomu však vykazuje vyšší výkon. MyISAM se hodí zejména pro tabulky, které slouží především pro čtení, a obecně pro menší projekty [3].

Dalšími zástupci jsou například Memory, který ukládá veškerá data v paměti RAM, dále CSV, který pro uložení dat využívá soubory ve stejnojmenném formátu, nebo Archive, sloužící pro ukládání velkého množství neindexovaných archivních dat [3].

2.1.1 MariaDB

MariaDB je nezávislá větev vyvíjená původními zakladateli MySQL, společností MySQL AB. Jedná se o open-source relační databázi, která se více zaměřuje na zapojení své komunity do vývoje [4].

MariaDB je z hlediska struktury databáze kompatibilní s MySQL, což znamená, že při přechodu z jednoho systému na druhý není potřeba upravovat databázové schéma. Oba systémy také využívají jazyka SQL, syntaxe je v obou případech identická. MariaDB se také oproti MySQL nesoustředí tolik na InnoDB jakožto hlavní engine, místo toho rozšiřuje možnosti o další enginy pro podporu různých specifických případů užití. MariaDB také podporuje datový formát JSON, ale na rozdíl od MySQL jej ukládá jako textový řetězec, nikoli v binární formě. Oba systémy tak nabízí odlišné funkce pro manipulaci s JSON dokumenty. MySQL však také podporuje MySQL Document Store, díky kterému mohou uživatelé ukládat JSON dokumenty do kolekcí místo běžných databázových tabulek a přistupovat k nim pomocí běžných CRUD operací namísto SQL.

MariaDB také nabízí některé nové mechanismy, jako jsou sekvence, neviditelné sloupce nebo sloupcové uložení dat. Oproti MySQL však postrádá podporu pro některé druhy indexů - sestupné, funkční nebo neviditelné (nevyužívané optimalizátorem).

Z hlediska výkonu a škálovatelnosti nabízí MariaDB tři nové funkce. Jedná se o MaxScale, který poskytuje load-balancing a umožňuje cachování výsledků vyhledávání v technologii Redis. Dále Spider, poskytující sharding a paralelizaci pro využití potenciálu více souběžných instancí nebo CPU. A nakonec Xpand, který zajišťuje distribuovanost a flexibilitu při přidávání nebo odebírání databázových instancí [4].

Název licence	**Cena**
MariaDB Enterprise Subscription	dostupné po kontaktování týmu MariaDB

Tabulka 2.3: Cena licence MariaDB Enterprise ke dni 10.2.2021 (dostupné na https://mariadb.com/pricing/)

Stejně jako MySQL, i MariaDB nabízí jak komunitní, tak Enterprise verzi, která navíc poskytuje vylepšený výkon díky query cache, zvýšení bezpečnosti a další doplňující nástroje. Cena Enterprise verze není pevně stanovená v ceníku a je potřeba kontaktovat společnost MariaDB pro získání konkrétní nabídky (viz tabulka 2.3).

Organizace MariaDB nabízí také cloudové řešení SkySQL pro zákazníky databáze MariaDB.

2.2 MongoDB

MongoDB je dokumentově orientovaná databáze, patří tedy do rodiny tzv. NoSQL databází. Ukládá data v podobě dokumentů ve formátu BSON, tedy binární JSON. Podporuje dynamické schéma a komplexní datové typy jako jsou pole, vnořené objekty nebo prostorová data [5].

Od verze 3.2 používá MongoDB jako své výchozí úložiště WiredTiger. To aplikuje pro většinu operací čtení a zápisu optimistický přístup řízení souběhu (verzování) na úrovni dokumentů a umožňuje tak v jednom okamžiku modifikaci více různých dokumentů od různých klientů [6].

Databáze MongoDB je distribuovaná, jelikož podporuje sharding, který rozděluje data na horizontální úrovni. Sharding slouží k rozdělení dat na více serverů, což snižuje výkonnostní a paměťové nároky na jednotlivé stroje.

Název služby	**Cena za hodinu**	**Cena za měsíc**
MongoDB Atlas	$1,77	$1274,4

Tabulka 2.4: Cena služby MongoDB Atlas u poskytovatele Amazon Web Services pro modelový příklad ke dni 10.2.2021 (vypočteno z https://cloud.mongodb.com/)

MongoDB je open-source, což znamená, že jej uživatelé mohou používat zdarma. Pokud se však uživatelé chtějí vyhnout pořizování a údržbě vlastních serverů, mohou využít technologii MongoDB Atlas, tedy nástroj pro vytvoření a správu MongoDB databáze v cloudovém úložišti. Cena modelového příkladu, úložiště s kapacitou 240 GB, 4 CPU a 32 GB paměti, je uvedena v tabulce 2.4.

2.3 Elasticsearch

Elasticsearch je distribuovaný vyhledávací a analytický engine, který umožňuje rychlé vyhledávání napříč různými typy dat. Je založen na technologii Apache Lucene. Elasticsearch poskytuje především dobrou škálovatelnost. K dispozici je také přidružený nástroj Kibana, který umožňuje uživatelům vizualizovat data, a Logstash, což je nástroj pro zpracování, transformaci a vkládání dat pocházejících z různých zdrojů. Tyto tři technologie se dohromady označují jako ELK Stack.

Název licence	**Cena za hodinu**	**Cena za měsíc**
Standard	$0,2152	$154,9
Gold	$0,2536	$182,6
Platinum	$0,2928	$210,8
Enterprise	$0,3512	$252,9

Tabulka 2.5: Ceny jednotlivých licencí Elasticsearch u cloudového poskytovatele Amazon Web Services pro modelový příklad ke dni 10.2.2021 (vypočteno z https://cloud.elastic.co/pricing)

Elasticsearch je open-source. Uživatelé jej mohou zdarma nainstalovat a používat, ale pro využívání cloudového úložiště je potřeba si zakoupit licenci. Ceny jednotlivých licencí pro modelový příklad (jediná instance, 8 GB RAM, 240 GB úložiště) jsou uvedeny v tabulce 2.5. S vyšší úrovní licence jsou uživatelům poskytovány například dodatečné funkce, zabezpečení nebo online podpora [7].

2.3.1 Klíčové pojmy

Index je v Elasticsearch kolekce *dokumentů* ve formátu JSON. Každý dokument představuje soubor klíčů (názvů polí) a jejich příslušných hodnot, které mohou mít různé datové typy. Na výběr jsou základní datové typy, jako jsou textové řetězce, numerické hodnoty, data nebo hodnoty typu boolean, ale také komplexní datové typy v podobě polí, objektů a vnořených objektů. Elasticsearch umožňuje taktéž práci s prostorovými daty.

Elasticsearch pracuje s tzv. *invertovanými indexy*, které umožňují velmi rychlé fulltextové vyhledávání. Jedná se o struktury, které obsahují všechna unikátní slova nalezená v daném indexu a seznam dokumentů, v nichž jsou tato slova přítomna.

V indexu často existuje velký objem dat, což může negativně ovlivňovat výkon. Z toho důvodu je možné rozdělit data v indexech na shardy, které horizontálně rozdělují data v indexu na více částí a podporují tak paralelizaci. Elasticsearch podporuje distribuovanost, k čemuž slouží vytváření *clusterů*. To jsou svazky uzlů (serverů), na kterých běží instance Elasticsearch. Díky tomu se zvyšuje kapacita a snižuje zátěž na jednotlivé uzly a shardy v těchto uzlech. Elasticsearch automaticky balancuje více uzlové clustery pro zvýšení výkonu a dostupnosti.

Ochrana proti ztrátě dat je v Elasticsearch zajištěna pomocí vytváření *replik*, které se chovají jako redundantní kopie datových shardů a poskytují jak zálohu pro případ hardwarového selhání, tak zvýšení výkonu při operacích čtení [8].

2.3.2 Analýza textu

Elasticsearch poskytuje funkcionalitu analyzátorů, které hrají klíčovou roli při fulltextovém vyhledávání (nad atributy typu `text`). Umožňují analyzovat a transformovat indexovaný text na různých úrovních. V rámci Elasticsearch existují výchozí (vestavěné) analyzátory, ale lze také nadefinovat vlastní analyzátory (vytvořené uživatelem) pro specifické potřeby. Ty se skládají ze tří částí. První částí je znakový filtr (*character filter)*, který umožňuje předzpracovávat stream znaků a přidávat, upravovat nebo odstraňovat z něj znaky. Znakové filtry mohou například odstraňovat z řetězců HTML elementy nebo nahrazovat specifické řetězce jinými řetězci.

Druhou částí je tokenizer, který rozděluje vstup na jednotlivé tokeny na základě zvoleného separátoru, nejčastěji nějakého whitespace znaku, ale obecně jakéhokoli znaku nebo řetězce. Existují tokenizery orientované na slova, které rozdělují text do jednotlivých slov podle prázdných znaků nebo nepísmenných znaků, dále tokenizery orientované na částečná slova, které vytvářejí ze slov N-Gramy a tokenizery orientované na strukturovaný text, které se obvykle používají spolu se speciálně strukturovaným textem, jako jsou emailové adresy, cesty k souboru, adresy a podobně.

Poslední částí jsou tokenové filtry, které mohou zvoleným způsobem transformovat vstupní tokeny předané z tokenizeru. Existuje opět více typů tokenových filtrů, může se jednat například o filtr pro převod textu na malá písmena nebo o odstranění tzv. stop slov (*stop words*). Stop slova jsou slova, která v daném jazyce nemají pro vyhledávání význam, jako jsou předložky, spojky nebo jiná běžná slova. Elasticsearch obsahuje seznam stop slov pro mnoho jazyků včetně češtiny. Následující příklad ukazuje vytvoření filtru pro česká stop slova:

```
PUT index_stop_words
{
  "settings": {
    "analysis": {
      "filter": {
        "czech_stop_words": {
          "type":      "stop",
```

```
          "stopwords": "_czech_"
        }
      }
    }
  }
}
```

Tokenové filtry mohou také měnit vstupní slova na jejich synonyma. Slova a jejich synonyma již nejsou zahrnuty v Elasticsearch a je tedy potřeba je definovat buď přímo v definici indexu, nebo ve speciálním konfiguračním souboru ve formátu Solr nebo WordNet. Následující příklad ukazuje vytvoření filtru pro česká synonyma s vlastním definovaným textovým souborem `czech_synonyms.txt` ve výchozím konfiguračním adresáři:

```
PUT index_synonyms
{
  "settings": {
    "analysis": {
      "filter": {
        "czech_synonyms": {
          "type":      "synonym",
          "synonyms_path": "czech_synonyms.txt"
        }
      }
    }
  }
}
```

Existují již hotová řešení pro česká synonyma, například v rámci OpenOffice. V této definici je však minimum definovaných synonym a jejich podoba je spíše symbolická. Pokud je tedy potřeba pro daný systém použít synonyma, je vhodnější připravit vlastní soubor synonym, který se vztahuje ke konkrétnímu případu užití a obsahuje relevantní synonyma pro tuto doménu [9]. Příkladem může být databáze seriálů, kde je vhodné definovat například synonyma pro různá označení sérií a epizod. Vyhledání *S01E01* by tak mělo vrátit stejný výsledek jako *01x01* a další alternativy.

Elasticsearch také nabízí koncept redukování slova na jeho kořen (dále *stemming*). Díky tomu je možné odhalit stejný kořen slova pro různé tvary indexovaných a vyhledávaných slov. Toho lze dosáhnout dvěma způsoby: algoritmicky nebo slovníkově. Algoritmický stemmer aplikuje na slova různá jazykově specifická pravidla. Výhodou je, že je rychlý a paměťově nenáročný, ovšem nemusí vždy fungovat správně, zejména u nepravidelných slov. Oproti tomu slovníkové stemmery využívají pro získání kořenu slova statický slovník. To, jak je stemming kvalitní, určuje kvalita vypracování

daného slovníku. Slovníkové stemmery jsou sice pomalejší a náročnější na paměť i přípravu, ale dokáží lépe pracovat s nepravidelnými slovy nebo slovy, které mají sice stejný kořen slova, ale jiný význam.

Elasticsearch poskytuje vlastní stemmery v podobě tokenových filtrů, a to jak algoritmické, tak slovníkové. Je ovšem možné využít i externích řešení. Jedním z příkladů je slovníkový stemmer založený na slovníku *hunspell* [10, 11].

Při vytváření indexů v Elasticsearch je také důležité dbát na vhodně zvolené mapování jednotlivých polí na správné datové typy, aby bylo pak možné je efektivně využívat při vyhledávání. Pokud není mapování explicitně uvedeno, zvolí Elasticsearch vlastní mapování, které nemusí být vždy korektní. Pozdější změna již není možná bez nutnosti znovu sestavení celého indexu [7]. V rámci mapování lze specifikovat jednak datové typy atributů, ale také analyzátory, které se pro jednotlivé textové atributy budou používat. Následující příklad ukazuje vytvoření vzorového indexu s jeho nastavením a mapováním:

```
PUT my_index
{
    "settings": {
        "index": {
            "analysis": {
                "analyzer": {
                    "my_analyzer": {
                        "filter": ["lowercase"],
                        ...
                    }
                }
            }
        }
    },
    "mappings": {
        "properties": {
            "product_name": {
                "type": "text",
                "analyzer": "my_analyzer"
            }
        }
    }
}
```

V tomto příkladu je možné vidět vytvoření vlastního analyzátoru s názvem `my_analyzer`, který využívá tokenový filtr `lowercase`, který převádí znaky do malých písmen, a další případné nastavení. Tento analyzátor je pak použit pro textové pole s názvem `product_name`. Takto lze specifikovat více různých analyzátorů a nadefinovat jejich použití pro různé pole dle potřeby.

Kapitola 3

Cache

Cache je paměťové úložiště, které ukládá výsledek předcházejících přístupů k datům. Účelem je umožnění efektivnějšího (rychlejšího) přístupu k těmto datům při budoucím dotazování.

Cache může být na straně klienta (tzv. *client-side cache*). Díky němu se může klient vyvarovat opakovaného přenosu stejných dat ze serveru, díky čemuž dochází k šetření přenesených dat. Druhým typem je cache na straně serveru (tzv. *server-side cache*). Tento typ se používá pro snížení počtu nákladných databázových operací při opakovaném dotazování na stejná data ze stran mnoha klientů.

Použití cachování může výrazně zvýšit výkon aplikace a dramaticky snížit dobu odezvy. Paměť cache se často využívá pro ukládání samotných objektů (například uživatelů nebo produktů), případně pro ukládání uživatelského sezení (*session*), čili provozních dat konkrétního uživatele. Na straně klienta se v tomto případě uloží pouze jeho session ID. Samotná data jsou uložena v cache na straně serveru, kde klíčem je session ID a hodnotou je pole daných provozních dat. V případě selhání webového serveru dojde k přepnutí na jiný server, který pouze z cache přečte uživatelská provozní data a pokračuje v provozování daného uživatelského sezení [12].

Paměť cache lze ovšem využít také pro ukládání výsledků databázového vyhledávání (časté například u katalogů). Díky tomu je eliminována nutnost pokládat identické dotazy databázi opakovaně a lze se vyhnout nadbytečným přístupům k databázi, která bývá obvykle nejpomalejším článkem celé aplikace. Principem je uložení sady výsledků konkrétního dotazu do cache pod určitým klíčem, kterým je typicky nějaká podoba vstupního dotazu (například jeho hash). Součástí klíče/dotazu mohou být také informace o použitém třídění výsledků vyhledávání nebo limit a offset výsledků pro stránkování. Tato práce se zabývá touto metodou cachování na straně serveru pro umožnění znovu využití výsledků databázového hledání.

3.1 Operace čtení

Při fyzickém čtení dat z databázové tabulky dochází k sekvenčnímu průchodu tabulkou, přičemž jsou procházeny veškerá data. Složitost této operace je O(n), kde n je počet záznamů v této struktuře.

Použití B-stromu, který se často používá jako datová struktura pro implementaci tabulky (shlukovaná tabulka) nebo indexu (obsahujícího ROWID příslušného indexovaného záznamu v tabulce), dosahuje operace čtení (s výjimkou rozsahového dotazu) složitosti O(log n). Dochází zde však k náhodným čtením, což je z hlediska propustnosti výrazně dražší operace, než je operace sekvenčního čtení.

Tyto nákladné fyzické přístupy je potřeba provádět při každém čtení. Pokud jsou však data uložena v hlavní paměti, je přístup k nim výrazně rychlejší. Například pokud je v cache uložen výsledek předchozího databázového dotazu pod určitým klíčem, má poté přístup k tomuto klíči a datům složitost O(1). Obecně mají logické operace 10 - 1000× větší propustnost než je propustnost diskových operací [13, 14].

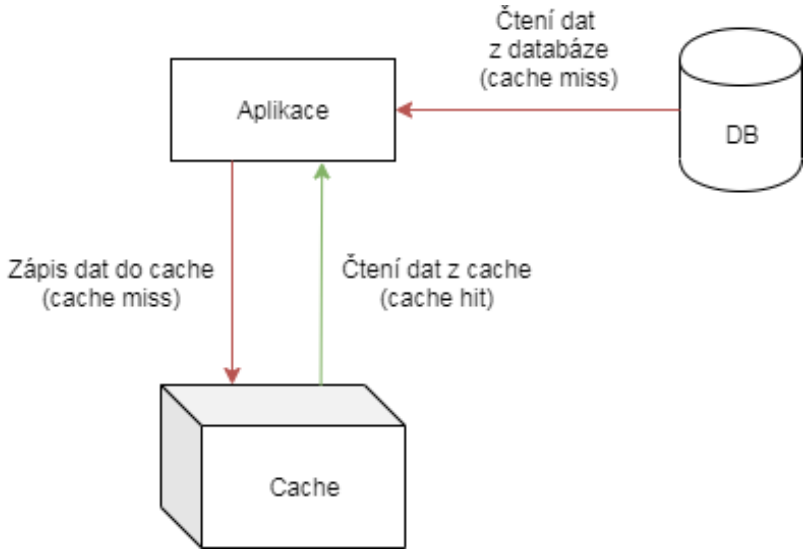
3.2 Statické a dynamické cachování

V aplikaci je možné provádět statické cachování, které je založeno na faktu, že četnosti opakování různých dotazů se liší. Toho lze využít a do cache vložit ty nejčastější dotazy a jejich výsledky. Tento přístup se nazývá statický, protože se cache takto naplňuje po dávkách a do příští aktualizace zůstává její obsah neměnný [15].

Oproti tomu u dynamické cache se může obsah za běhu měnit. Obecný princip dynamického cachování (také tzv. *Read through* nebo *Lazy loading* princip cachování) je následující:

1. Aplikace zkontroluje cache.
2. Pokud jsou hledaná data nalezena v cache, situace se nazývá *cache hit*. Data jsou přečtena z cache a vrácena do klienta.
3. Pokud data nejsou nalezena v cache, situace se nazývá *cache miss*. V tomto případě se aplikace musí na data dotázat samotné databáze a poté je uložit do cache, aby byla dostupná pro příští čtení.

Obrázek 3.1 ukazuje takto popsané schéma.



Obrázek 3.1: Princip fungování dynamické cache

Bylo prokázáno, že statické cachování je vhodná volba zejména u (velmi) malých cache, zatímco u větších cache je možné dosáhnout vyššího výkonu za použití dynamického přístupu [16].

3.3 Cache Hit Ratio

Hlavním ukazatelem efektivity fungování cache je tzv. *cache hit ratio*. Jedná se o poměr úspěšných vyhledávání v cache (cache hit) ku celkovému počtu dotazů na cache. Udává se v procentech. Snahou je dosáhnout co nejvyššího hit ratio. Pokud je velikost paměti cache malá, je možné do ní uložit pouze omezené množství dat. Kvůli tomu pak dochází častěji ke cache miss a cache hit ratio se blíží 0 %. Naopak čím více dat je uloženo v cache, tím více se cache hit ratio blíží 100 %.

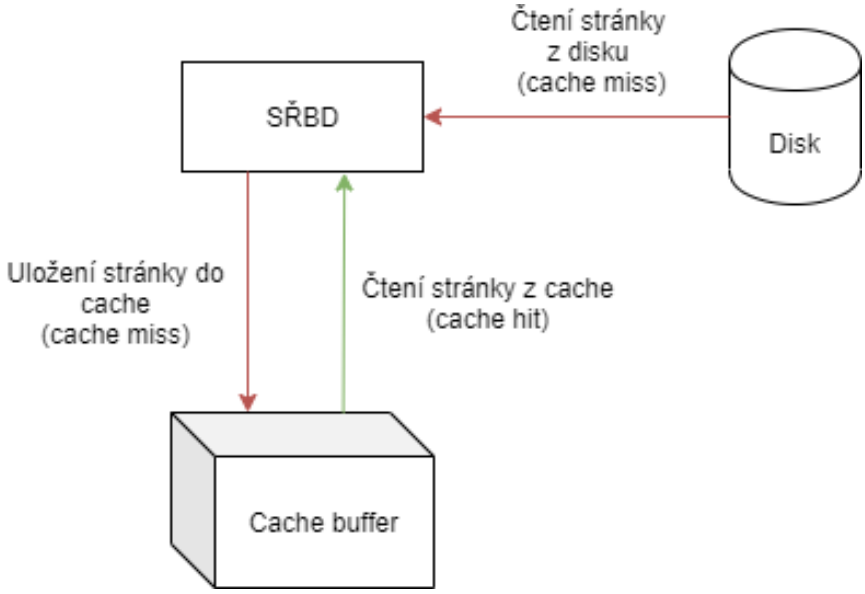
Existuje přímý vztah mezi velikostí cache a tímto poměrem - čím více výsledků vyhledávání je uloženo v cache, tím vyšší je šance, že tam budou data při příštím vyhledávání nalezena. V teoreticky nekonečně velkých cache pak tedy dochází ke cache miss pouze v nutných případech, tedy při prvním položení konkrétního dotazu [17]. Hlavním problémem se zde však stává zastaralost dat uložených v cache. Některé techniky, které poskytují řešení tohoto problému, jsou rozebrány v kapitole 5.

V opačném případě, tedy když pro cache není v daném systému dostatek místa v paměti, je potřeba využít některé mechanismy pro správu cache: vstupní politiku a uvolňovací politiku (viz kapitola 4).

3.4 Cache buffer

Snahou SŘBD je umístit maximální objem dat do hlavní paměti (tzv. *cache buffer*), čímž kompenzují obvykle pomalé přístupy na pevný disk a zvýšují tak propustnost databáze. Ideální hranicí

množství diskových stránek, které jsou dotazovány a nalezeny v buffer cache, je 90 %, což znamená, že maximálně u 10 % dotazovaných stránek dojde ke cache miss a bude je nutné načíst z disku.



Obrázek 3.2: Princip buffer cache.

Obrázek 3.2 ukazuje schéma fungování cache buffer. Při požadavku na provedení nějaké CRUD operace nad konkrétní stránkou nejprve SŘBD vyhledá tuto stránku v cache bufferu. Pokud zde není nalezena, načte tuto stránku z disku a uloží ji do paměti.

Cache buffer představuje frontu diskových stránek, přičemž první stránka v této frontě je ta, která byla využita jako poslední. Pokud je cache buffer zaplněn a je potřeba načíst novou stránku z disku do paměti, dojde nejprve k uložení první stránky ve frontě zpět na disk, díky čemuž se v paměti uvolní místo. Aktuálně používaná stránka se pak uloží na konec fronty. Tento princip uvolňování cache se nazývá LRU, viz kapitola 4.2 [14].

3.5 Query cache

Některé databáze (například MariaDB [18]) nabízí také vlastní integrovanou cache (tzv. *query cache*), kterou databázový systém sám spravuje a aktualizuje. Na rozdíl od cache buffer tato cache neukládá diskové stránky, ale již konkrétní data jakožto výsledek předchozího databázového vyhledávání. Výhodou této cache je to, že se uživatel o cachování nemusí starat a implementovat vlastní cache mechanismy. Integrovaná cache však přináší řadu nevýhod, zejména velikost této cache, která je limitována. Data v této cache navíc není možné sdílet s dalšími instancemi nebo uzly, což může být velmi nevýhodné [19]. Právě kvůli nevýhodám spojených s výkonem a špatnou škálovatelností byla například z databázového systému MySQL od verze 8.0 integrovaná cache (query cache) zcela odstraněna [20].

3.6 Distribuovaná cache

Tradiční koncept cache jako samostatné paměti v jednom uzlu je u dnešních systémů často rozšířen na tzv. distribuovanou cache, která je rozmístěna na více serverech. Díky tomu může narůst paměťová kapacita a výkon. Jedná se o horizontální škálování. Při použití distribuované cache lze s výhodou využít přidávání nových uzlů za běhu. Zároveň poskytuje vysokou dostupnost a toleranci chyb. Principem je využití tzv. clusteru, tedy shluku serverových uzlů, mezi které jsou data (obvykle automaticky na základě klíče či hashe) rozdělena [12].

Následující příklad ukazuje fungování populární technologie pro distribuované cachování - Redis Cluster. Jedná se o distribuovanou implementaci paměťového úložiště Redis. Je dána hashovací tabulka, která obsahuje určitý počet slotů (konkrétně 16384) a hashovací funkce, která každý klíč jednoznačně zařadí do jednoho z těchto slotů. Každý uzel v clusteru je pak zodpovědný za určitou podmnožinu těchto slotů.

Pokud jsou v systému přítomny například 3 uzly, bude první uzel obsahovat sloty 0-5500, druhý uzel sloty 5501-11000 a poslední uzel sloty 11001-16383. Ukázka fungování Redis Cluster z webu redis.io [21] se třemi sloty je uvedena ve výpisu 3.1. Je zde vidět, že ať je klíč nastavován nebo získáván z jakéhokoli uzlu, dojde vždy k přesměrování na správný uzel, který daný klíč drží.

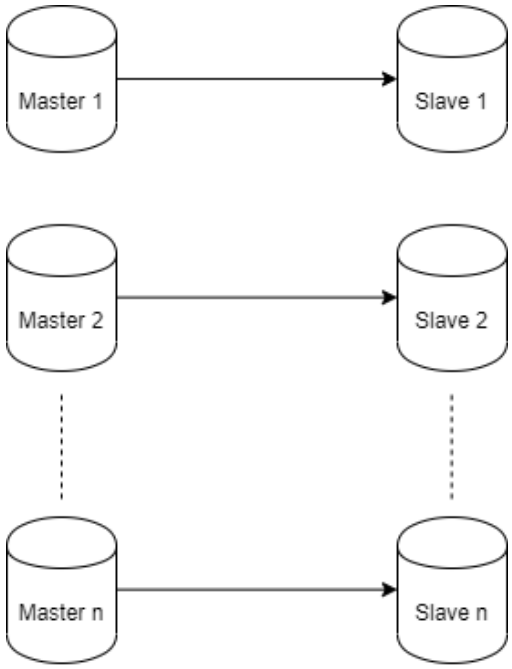
```
$ redis-cli -c -p 7000
redis 127.0.0.1:7000> set foo bar
-> Redirected to slot [12182] located at 127.0.0.1:7002
OK
redis 127.0.0.1:7002> set hello world
-> Redirected to slot [866] located at 127.0.0.1:7000
OK
redis 127.0.0.1:7000> get foo
-> Redirected to slot [12182] located at 127.0.0.1:7002
"bar"
redis 127.0.0.1:7002> get hello
-> Redirected to slot [866] located at 127.0.0.1:7000
"world"
```

Výpis 3.1: Ukázka fungování Redis Cluster (příklad z webu redis.io)

Pro zajištění dostupnosti i v případě selhání některých uzlů se využívá princip master-slave. Každý slot v hashovací tabulce má 1 až N svých kopií, které jsou uloženy jednak v uzlu typu master a poté v N-1 uzlech typu slave. V případě, že jeden z master uzlů přestane odpovídat, cluster vyhledá jeho kopii (uzel typu slave) a povýší ho na nový uzel typu master. Systém tak může dál fungovat.

Vzhledem k tomu, že se jedná o asynchronní replikaci, může za určitých podmínek dojít ke ztrátě aktualizací dat [13].

Následující obrázek 3.3 ukazuje jednoduché schéma Redis Cluster s N uzly typu master a vždy jedním uzlem typu slave ke každému uzlu typu master.



Obrázek 3.3: Schéma Redis Cluster

Kapitola 4

Plnění a uvolňování cache

Vzhledem k tomu, že cache jsou paměťově omezená úložiště, je vhodné zajistit některé mechanismy pro správu jejich obsahu.

4.1 Vstupní politika

Ačkoli může cachování výrazně zvýšit výkon aplikace, ne vždy musí být výhodné ukládat do cache všechny dotazy a jejich výsledky. Zejména u systémů, ve kterých je pro cache k dispozici málo paměti a je proto potřeba jí šetřit, je možné identifikovat dotazy, jejichž výsledky není potřeba ukládat do cache, protože se s nejvyšší pravděpodobností nebudou (často) opakovat. Díky tomu je možné ušetřit cenné místo v paměti pro výsledky obecnějších vyhledávání, které se mohou v budoucnu objevovat častěji, a zvyšovat tím cache hit ratio.

Odhad míry, do jaké je dotaz opakovatelný, lze provádět staticky nebo dynamicky. Statické odhady se zaměřují na kvality dotazu, jako je jeho délka nebo množství vyhledávání slov ve full-textovém vyhledávání. Obvykle platí, že čím delší je dotaz (z hlediska počtu predikátů nebo délky vyhledávaného textu), tím menší je pravděpodobnost, že se tento dotaz bude opakovat. Jedním z možných řešení je implementace modulu, který tímto způsobem na základě předepsaných pravidel pro každý dotaz vyhodnocuje, zda se vyplatí jej ukládat do cache, nebo nikoli. Dynamické odhady se pak odvíjejí od analýz a statistik pokládaných uživatelských dotazů [22].

4.2 Uvolňování cache

V případě naplnění paměti může být potřeba z cache odstranit některé záznamy, aby mohly být vloženy nové. Existuje více způsobů, jakými určit, který záznam má být odstraněn. Cílem tohoto mazání by mělo být především nahrazení méně využívaných hodnot, zatímco často využívané hodnoty by měly v cache zůstat, aby nedošlo ke snížení cache hit ratio.

Většina cache technologií nabízí různé algoritmy (*eviction policies*), které umožňují automaticky mazat nebo nahrazovat některé záznamy v cache. Konkrétně zvolený algoritmus pak záleží na povaze aplikace.

Následující seznam shrnuje nejobvyklejší algoritmy, které se používají pro nahrazení záznamů v cache [23]:

- Least Recently Used (LRU) – nahrazuje se záznam, který nebyl využit po nejdelší dobu. Jedná se o nejpoužívanější algoritmus v cachování.
- Least Frequently Used (LFU) – nahrazuje se záznam, který byl využit nejméně krát. Tento způsob tedy bere v potaz frekvenci dotazování na jednotlivé klíče.
- Most Recently Used (MRU) – nahrazuje se naposledy využitý záznam. Tento způsob je vhodný u aplikací, u kterých předpokládáme, že po využití nějaké hodnoty tato hodnota již (alespoň v nejbližší době) nebude znovu využita.
- First in first out (FIFO) – záznamy se nahrazují v pořadí, v jakém byly vloženy, bez ohledu na četnost nebo frekvenci přístupů k nim. Algoritmus využívá princip zásobníku.
- Random replacement (RR) – nahrazuje se náhodně vybraný záznam.

Záznamy mohou být z cache odstraňovány i za jinými účely než z důvodu uvolnění paměti. Hlavním takovým důvodem je zajišťování integrity a aktuálnosti dat. Tato problematika je popsána v kapitole 5.

4.3 Deduplikace vstupních dotazů

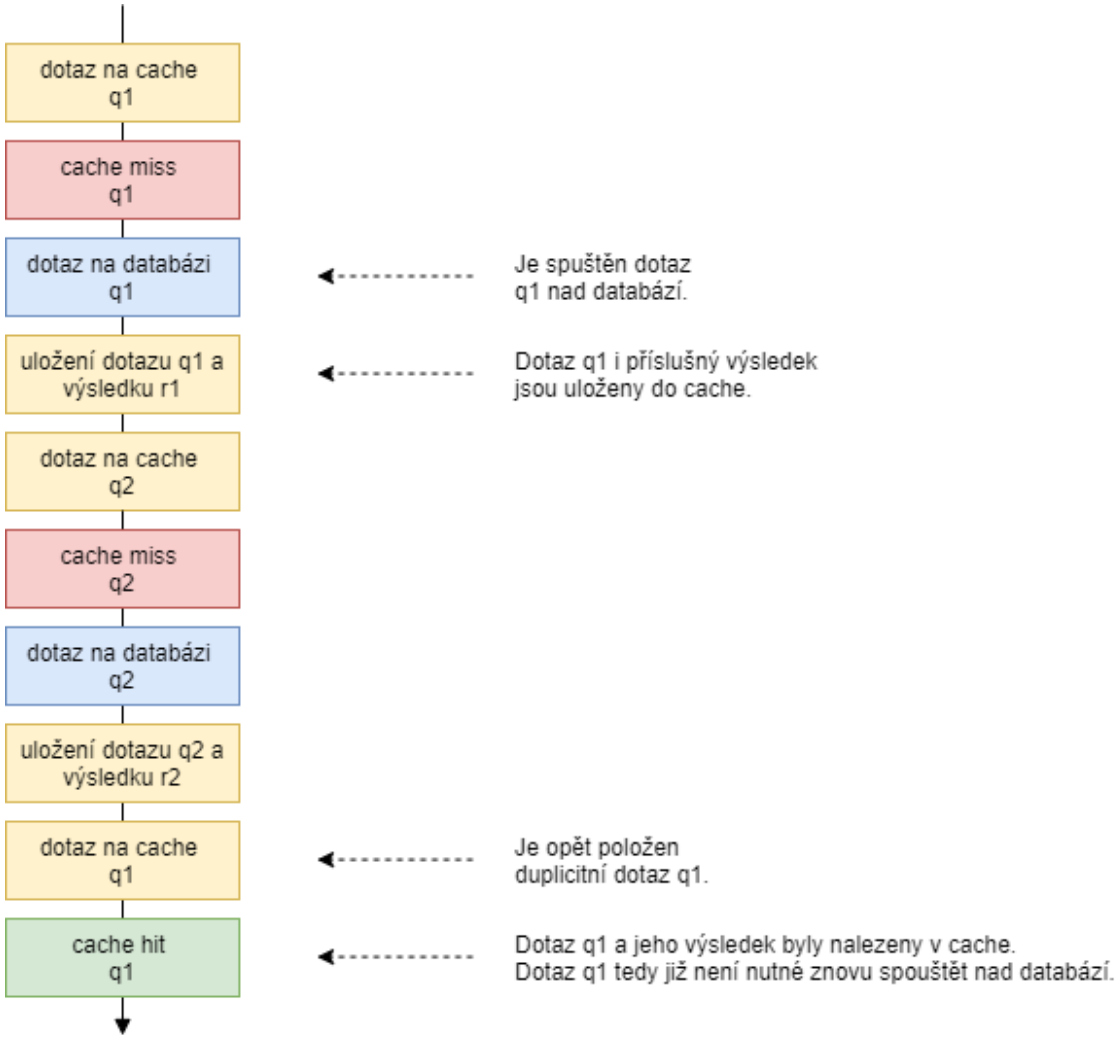
Princip ukládání výsledků databázového hledání spočívá v uložení těchto výsledků do cache pod určitým klíčem. Tímto klíčem by měl být například hash samotného dotazu. Při příštím dotazu na data se opět vytvoří hash tohoto nového dotazu a nejprve se zkontroluje cache, jestli již byl tento hash uložen.

Tento postup vede k tomu, že pokud mají dva dotazy jen bezvýznamnou odlišnost, jako jsou přidané bílé znaky, jiné pořadí predikátů nebo text lišící se ve velikosti písmen, budou mít rozdílný hash a budou tak identifikovány jako odlišné. Pro eliminaci tohoto problému je možné provést určité normalizace dotazu, jako je odstranění bílých znaků, převod dotazu do malých písmen a podobně.

Výhodou také je, že webové aplikace často využívají systém připravených parametrizovaných šablon pro SQL dotazy, do kterých jsou posléze dosazovány parametry v závislosti na uživatelském hledání. Díky tomu je omezen počet možných kombinací a podob vzniklých dotazů [24, 25].

Obrázek 4.1 ukazuje princip deduplikace vstupního dotazu. V tomto schématu je nejprve položen dotaz *q1*. Jelikož není nalezen v cache, musí být spuštěn nad databází. Dotaz a jeho výsledek je pak uložen do cache. Následuje dotaz *q2*, u kterého opět dochází ke cache miss, tudíž se postup opakuje.

Poté je však znovu položen dotaz *q1*, který je identifikován jako duplikát dříve položeného dotazu *q1* a je již v cache nalezen. Díky tomu se nemusí provádět drahá operace dotazování nad databází. Místo toho je navrácen dříve uložený výsledek z cache.



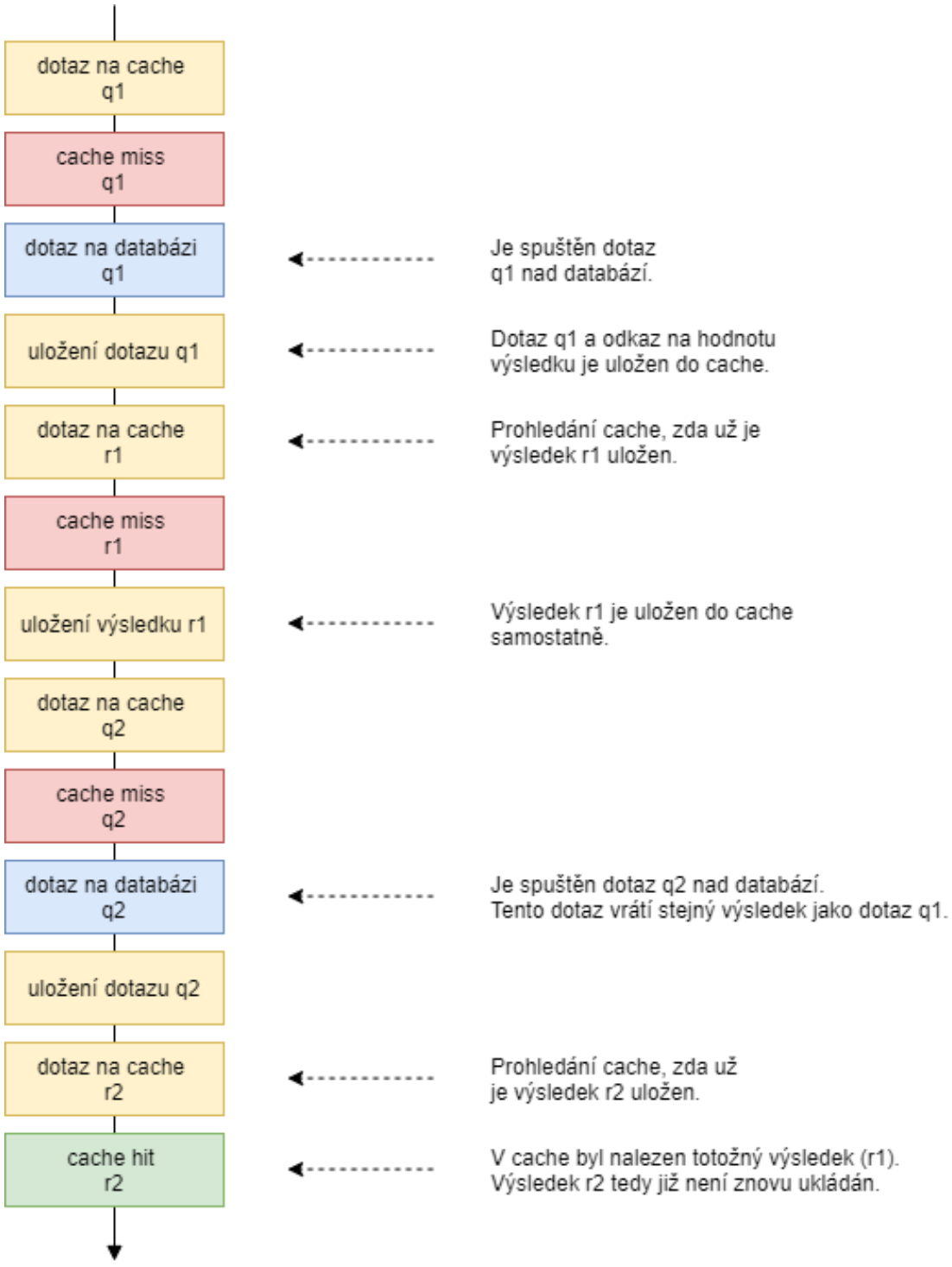
Obrázek 4.1: Princip deduplikace vstupního dotazu

4.4 Identifikace dotazů se stejnými výsledky

Při cachování výsledků databázového hledání mohou být ukládané hodnoty reprezentující výsledky dotazů velmi velké. Proto pokud více dotazů vrací stejnou sadu záznamů, je výhodné tento výsledek v cache uložit pouze jednou a následně se na něj odkazovat.

Toho lze dosáhnout opět pomocí hash hodnot. Výsledek databázového hledání se neuloží do cache přímo pod klíčem pro daný dotaz, ale pod samostatným klíčem, kterým je hash tohoto výsledku.

Hodnotou pro daný klíč (představující databázový dotaz) pak bude odkaz na tento výsledek. Díky tomu bude každý unikátní výsledek hledání uložen v cache pouze jednou.



Obrázek 4.2: Princip deduplikace výsledků vyhledávání na základě různých vstupních dotazů

Obrázek 4.2 ukazuje princip deduplikace výsledků vyhledávání na základě různých vstupních dotazů. Na tomto schématu lze nejprve vidět položení dotazu *q1*, který není nalezen v cache, tudíž musí být spuštěn nad databází. Dotaz je následně uložen do cache a odkazuje na hodnotu *r1* (hodnotou je tedy hash výsledku *r1*). Dále je provedeno hledání, zda se již taková hodnota v cache nenachází. Teprve když tato hodnota v cache není nalezena, je výsledek r1 uložen samostatně do cache. Následně je položen jiný dotaz *q2*, jehož výsledek *r2* je však totožný jako výsledek *r1*. Při hledání je výsledek se stejnou hash hodnotou již nalezen. Výsledek *r2* tedy není znovu ukládán do cache, místo toho bude dotaz *q2* rovněž odkazovat na výsledek *r1*.

Tento přístup je zvlášť výhodné kombinovat s fulltextovými vyhledávači typu Elasticsearch, kde může mít vyhledávaný výraz mnoho podob, ale vést ke stejnému výsledku (viz kapitola 2.3.2).

Kapitola 5

Problém aktualizace dat

Cachování výsledků databázového hledání s sebou přináší také určité problémy. Zejména se jedná o problém, který představují aktualizace dat v databázi a s nimi spojené znehodnocování (*invalidation*) zastaralých dat v cache. Po aktualizaci dat v databázi mohou v cache zůstávat zastaralá data. Z toho důvodu je důležité v aplikaci poskytnout mechanismy, které zajistí aktualizaci dat v cache.

5.1 Expirace (TTL)

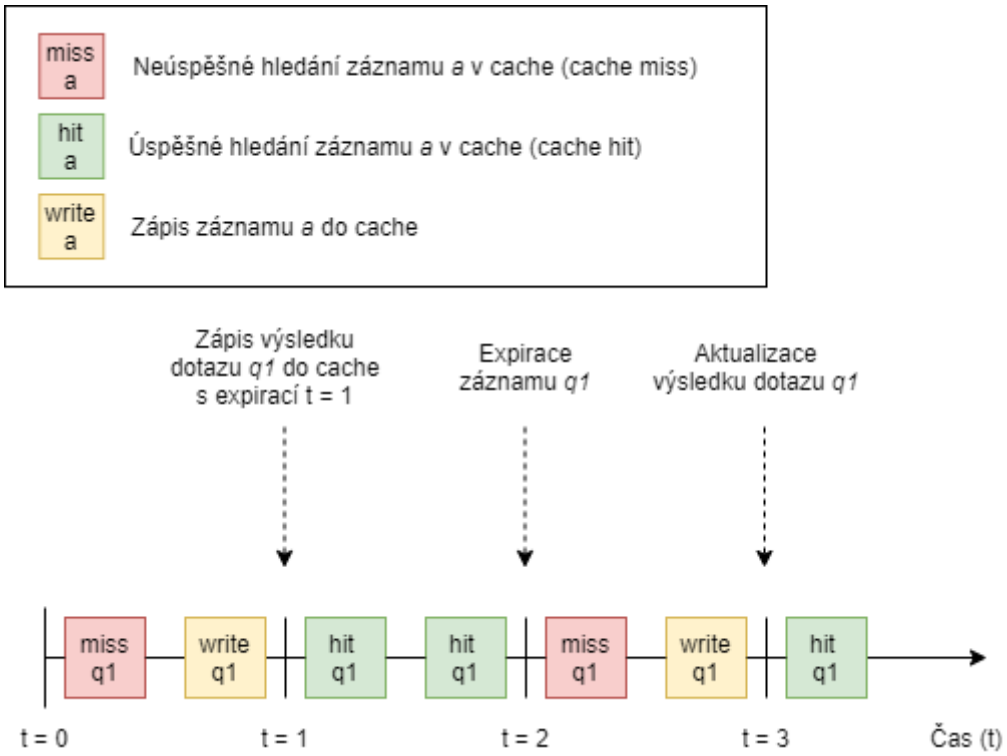
Častým řešením je nastavování *time to live (TTL)*, kdy se každé cachované hodnotě nastaví určitý fixní čas, po jehož uplynutí přestane být hodnota považována za validní a bude se muset aktualizovat. Díky tomu je zaručeno, že doba existence potenciálně zastaralé hodnoty v cache nepřekročí definovanou horní hranici. Hodnota time to live by měla odpovídat povaze systému – při příliš nízkých hodnotách by muselo docházet k častému provádění nákladné operace dotazování se nad databází, naopak při příliš vysokých hodnotách by se zvyšoval čas, po který jsou uživatelům zobrazována potenciálně zastaralá data. Používání pouze TTL je tedy metoda, která je vhodná v aplikacích, u kterých dočasná práce se zastaralými daty nepředstavuje problém.

Obrázek 5.1 ukazuje, jak se nastavuje TTL u záznamu v databázi Redis. V prvním příkazu je záznamu nastaven TTL (*EX*) s hodnotou 10, tedy 10 vteřin. Následně je proveden dotaz na tento klíč a navrácena jeho hodnota. O 10 vteřin později je dotaz zopakován, ale tentokrát je již vrácena speciální hodnota *nil*, což znamená, že TTL záznamu už vypršel a hodnota již není k dispozici.

```
127.0.0.1:6379> SET klic hodnota EX 10
OK
127.0.0.1:6379> GET klic
"hodnota"
127.0.0.1:6379> GET klic
(nil)
```

Obrázek 5.1: Příklad nastavení TTL na 10 vteřin (Redis)

Diagram na obrázku 5.2 ukazuje princip expirace záznamu v cache.



Obrázek 5.2: Princip TTL (expirace) v cache

Je možné také implementovat doplňující algoritmus, který u záznamů s vypršenou platností znovu zavolá příslušný dotaz a data v cache obnoví. Nevýhodou však je, že se zvýší objem dotazů, které jsou prováděny zbytečně, aniž by pak nalezly nějaké využití [26].

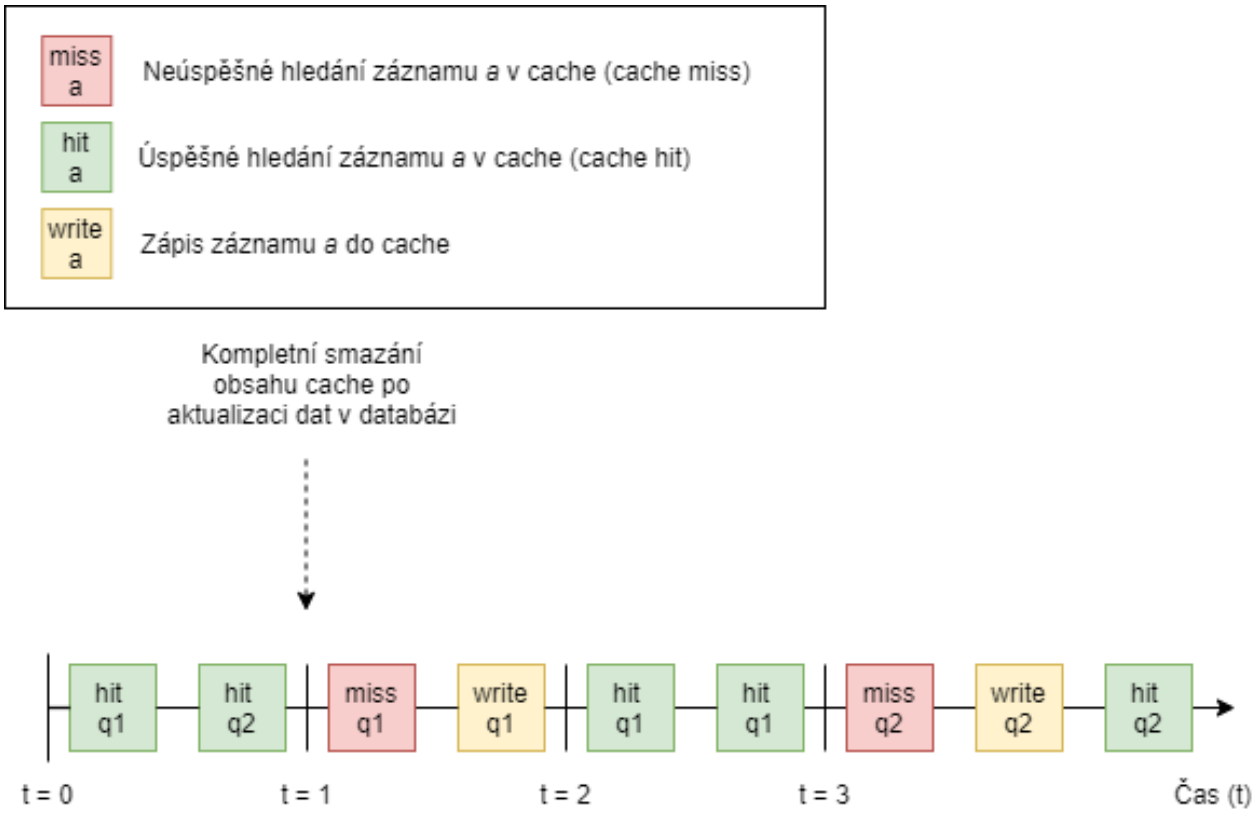
Často se v informačním systému nacházejí jak data, u kterých je potřeba zajistit vysokou úroveň konzistence, tak data, u kterých riziko zastaralosti nepředstavuje problém. Je proto možné nastavovat různým klíčům různé hodnoty TTL, případně některé kritické dotazy a jejich výsledky do cache vůbec neukládat. Například v rámci aukčního webového portálu mohou být do cache s vyšší hodnotou TTL uloženy výsledky databázových dotazů na bestsellery nebo poslední vydané články o produktech. Naopak data týkající se aktuálních nabídek u konkrétních produtků nebo

stavu skladových zásob by měly být ukládány s velmi krátkým TTL, případně by vůbec neměly být cachovány [25].

5.2 Smazání obsahu cache po aktualizaci

Jiným způsobem řešení tohoto problému je kompletní smazání cache při každé provedené aktualizaci dat. Postup je takový, že se při každé provedené změně v datech (z jakéhokoli zdroje) volá metoda, která obsah cache vymaže. Výhodou tohoto postupu je, že je velice rychlý a účinný. Ovšem dojde tak ke ztrátě veškerých záznamů v cache, a to i těch, kterých se změna v datech reálně nedotkla. Při každém následujícím volání nového dotazu tak dojde ke cache miss a bude nutné provést drahou operaci dotazování se nad databází.

Diagram na obrázku 5.3 ukazuje princip smazání veškerých záznamů v cache po aktualizaci v datech.

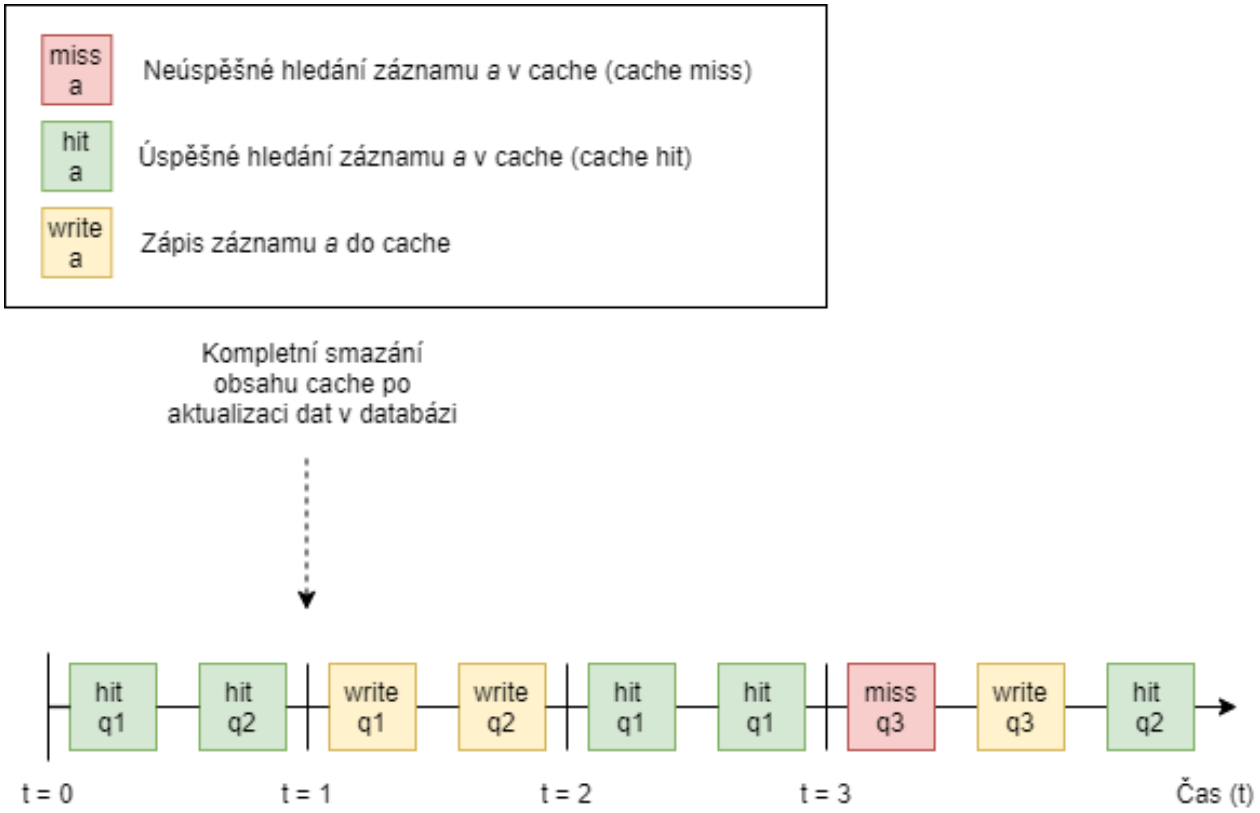


Obrázek 5.3: Princip smazání celého obsahu cache po aktualizaci dat v databázi

Tento způsob je možné s výhodou využít u systémů, u kterých dochází k aktualizacím jen velmi zřídka a kompletní smazání cache (a následné znovu uložení hodnot) není příliš nákladné. Naopak v případě, že dochází ke změnám v databázi často, je tento postup nevhodný, protože kvůli častému mazání obsahu cache zaniká efektivita cachování.

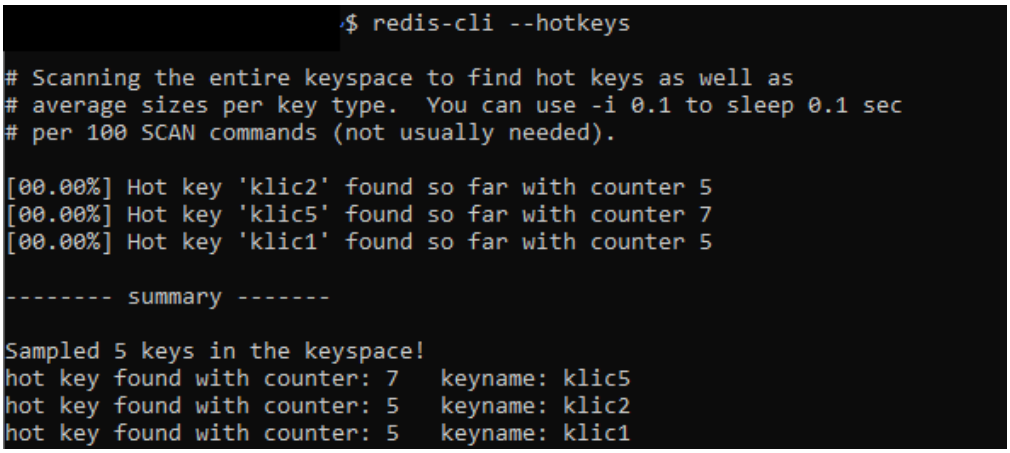
5.3 Předběžné cachování častých dotazů

Určitým kompromisem je sice smazat celou cache, ale následně ručně cachovat často vyhledávané hodnoty. Tento princip je ukázán na obrázku 5.4.



Obrázek 5.4: Princip předběžného cachování častých dotazů po smazání obsahu cache

Je několik způsobů, jak zjišťovat, které klíče jsou dotazovány nejčastěji. Například samotný Redis od verze 4.0 nabízí příkaz OBJECT FREQ, který vrací u daného klíče logaritmickou četnost jeho dotazování [13]. Je také možné využít parametr –hotkeys [27], jak ukazuje obrázek 5.5:



Obrázek 5.5: Použití –hotkeys v databázi Redis

Případně je možné implementovat vlastní počítadlo, které při každém dotazu na klíč připočítá hodnotu 1 k danému klíči do oddělené struktury v cache nebo databázové tabulky. Díky tomu je možné ukládat si konkrétní počty dotazů na jednotlivé klíče. Ihned po smazání obsahu cache je pak možné určit sadu typických dotazů s nejvyššími četnostmi dotazování. Pro znovu sestavení cache stačí cyklem projít každý z daných klíčů a sestavit z něj (nebo dohledat) původní dotaz. Ten následně automaticky položit databázi a výsledek už standardně uložit do cache. Tento postup je implementován v testovací aplikaci, viz kapitola 7.

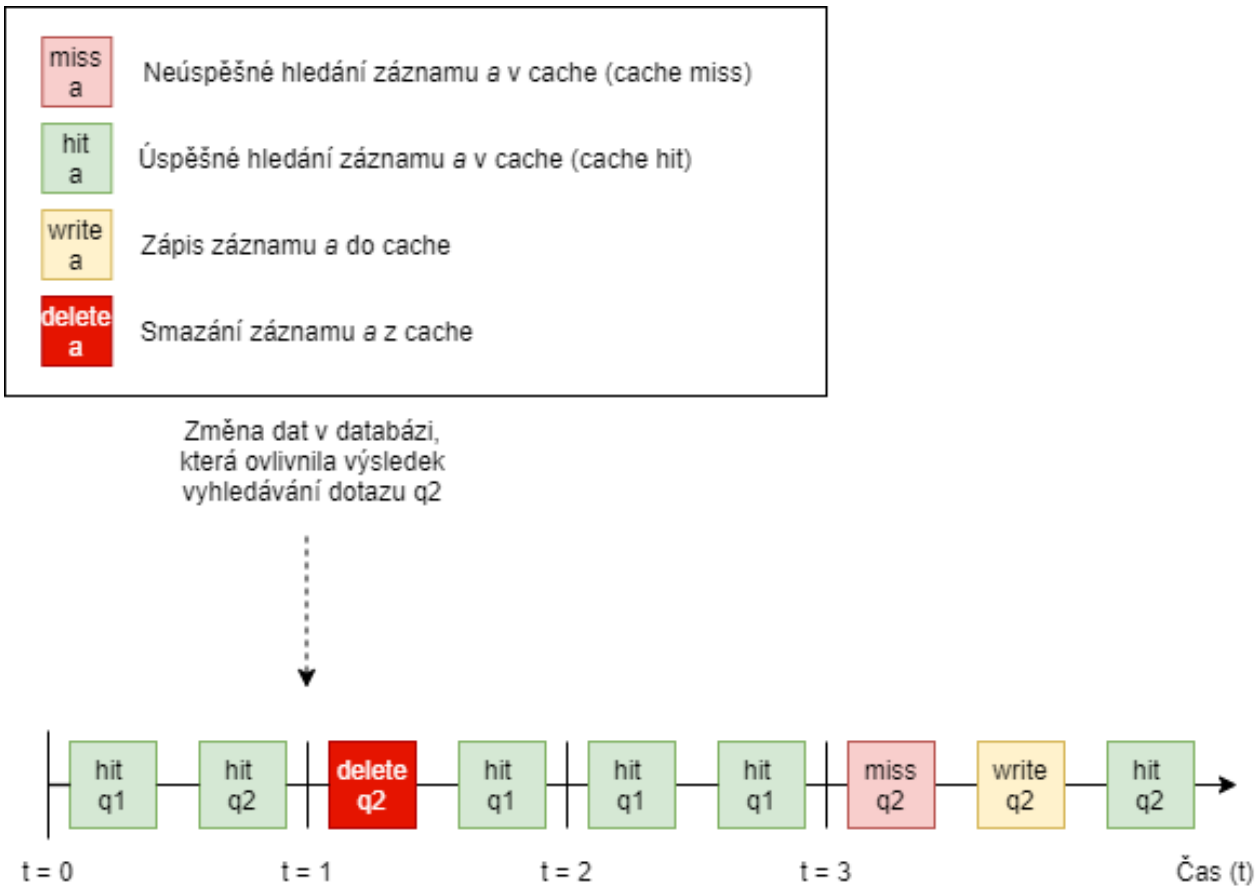
5.4 Využití paralelních instancí

U velkého množství dotazů může znovu sestavování cache nějakou dobu trvat. Pokud je potřeba se vyhnout tomuto čekání, je možné mít k dispozici dva paralelní stroje/uzly, z nichž na jednom běží aktivně používaná produkční databáze. Na druhém stroji běží pracovní verze databáze, na které se provádí aktualizace dat, sestavování indexů a případné znovu sestavování cache pro typickou sadu dotazů. V určité chvíli se pak nastaví produkční databáze na tuto aktualizovanou, nově sestavenou databázi a původně produkční databáze se odstaví. Tyto dva nebo více strojů se takto neustále střídají ve stanovených časových intervalech.

Tento postup je výhodný použít například u produktových srovnávačů, které průběžně stahují data z různých internetových obchodů a aktualizují cenové nabídky a údaje o jednotlivých produktech. Tyto změny není nutné ihned promítat do produkční databáze, jejíž výkon by byl tímto omezen. Naopak je možné provádět aktualizace na pozadí na odstaveném pracovním stroji a změny promítnout do produkční databáze až po uplynutí určitého časového intervalu.

5.5 Smazání pouze relevantních záznamů

Dalším řešením je vymazat z cache pouze záznamy, které se vztahují k aktualizovaným datům. Díky tomu nedojde ke ztrátě dat, která nesouvisejí s upraveným nebo vloženým záznamem. Tento princip je znázorněn na obrázku 5.6.



Obrázek 5.6: Smazání pouze relevantních záznamů v cache po aktualizaci dat v databázi.

Pro toto řešení však musí existovat relativně složitý algoritmus, který pro každý klíč umí vyhodnotit, zda se vztahuje k nové hodnotě a zda je tedy nutné jej smazat. Je totiž nutné brát v potaz tyto scénáře:

1. Byl vložen nový záznam. Je potřeba znehodnotit klíče obsahující výsledky vyhledávání, které se vztahují k atributům nově vloženého záznamu.

2. Byl upraven záznam. Je potřeba vyhodnotit, který atribut byl upraven, a znehodnotit výsledky vyhledávání, které se vztahují jak k původní hodnotě atributu (záznam již nebude součástí výsledku tohoto vyhledávání), tak k nové hodnotě atributu (záznam bude nyní součástí výsledků jiných vyhledávání).

3. Byl odstraněn záznam. Je potřeba odstranit z cache výsledky vyhledávání, které obsahovaly tento záznam.

Tento algoritmus by mohl mít vysokou složitost jak na implementaci, tak na vykonání. Implementace by mohla být taková, že by se zvlášť ukládal seznam objektů, které daný dotaz vrací. Při aktualizaci (nebo smazání) daného objektu by pak došlo ke znehodnocení všech dotazů, které tento objekt obsahovaly. Tento postup však neřeší problém znehodnocení záznamů, které aktualizovaný (či vložený) objekt neobsahovaly, ale obsahovat nově mají. Z toho důvodu by bylo stejně potřeba jednou za stanovený čas cache promazat a znovu sestavit. Navíc je tento způsob paměťově náročný, protože je zde nutné kromě samotných výsledků vyhledávání ukládat zvlášť ještě seznamy objektů pro každý dotaz. Vzhledem k nutnosti vyhledávání a spravování dalších paměťových struktur pak není možné zaručit, že tento postup nebude kontraproduktivní, tedy dokonce více časově náročný než přístup do databáze [28].

Tento postup je také velmi náchylný na chyby. V případě, že by v aplikaci byl implementován tento algoritmus jako jediná metoda znehodnocování dat v cache a došlo by jeho prováděním k chybě (ať už k pádu systému či pouze k sémantické chybě při vyhodnocování záznamů, které mají být smazány), hrozilo by, že by zastaralá data zůstala v cache po velmi dlouhou dobu a změny v datech by se tak správně nepromítnuly do aplikace.

Další nevýhodou je nutnost připravit mechanismus, který provádí určitou inspekci cachovaných dat, aby byl schopen rozhodnout, zda je potřeba tato data znehodnotit. To může představovat problém z hlediska bezpečnosti, protože tento přístup nedovoluje například práci s šifrovanými daty [25].

V mnohých případech je však možné data nějakým způsobem kategorizovat - například v katalogu zboží se obvykle vyskytuje mnoho různých kategorií jednotlivých produktů. V případě, že je do cache současně s výsledkem vyhledávání uložena také hodnota identifikující kategorii, pod kterou dané zboží spadá, je možné při změně v datech odstranit pouze klíče, které spadají pod postiženou kategorii a nedojde tak ke kompletní ztrátě všech cachovaných dat.

5.6 Shrnutí

Z hlediska znehodnocování dat v cache neexistuje žádný ideální přístup, který by byl výhodný ve všech případech. Každý přístup má svá omezení a je nutné zvážit, zda nebudou tato omezení překážkou pro daný informační systém.

Tabulka 5.1 srovnává základní vlastnosti jednotlivých přístupů. Ukazuje zejména zda daný přístup podporuje okamžité promítnutí změn. To je velmi důležitý ukazatel, protože u některých informačních systémů je okamžité promítnutí změn zcela klíčové a jakákoli prodleva by mohla mít negativní následky. U většiny systémů však případná prodleva nepředstavuje závažné problémy. Druhým ukazatelem je efektivita, tedy zda je plně využit potenciál paměti cache. U přístupů, kde je

vysoká efektivita cache, nelze zaručit okamžité promítnutí změn, a naopak v případě, že je potřeba zaručit okamžité promítnutí změn v datech, není cache příliš efektivní a je potřeba zvážit, zda je vůbec vhodné její použití. Posledním ukazatelem je náročnost na implementaci a odolnost proti chybám.

Přístup	**Okamžité promítnutí změn**	**Efektivita cache**	**Náročnost implementace**
Expirace	NE	ANO	NE
Smazání obsahu cache po aktualizaci	ANO	NE	NE
Předběžné cachování častých dotazů	ANO	ČÁSTEČNĚ	ČÁSTEČNĚ
Využití paralelních strojů	NE	ANO	ČÁSTEČNĚ
Smazání pouze relevantních záznamů	ANO	ČÁSTEČNĚ	ANO

Tabulka 5.1: Srovnání přístupů ke znehodnocování dat v cache.

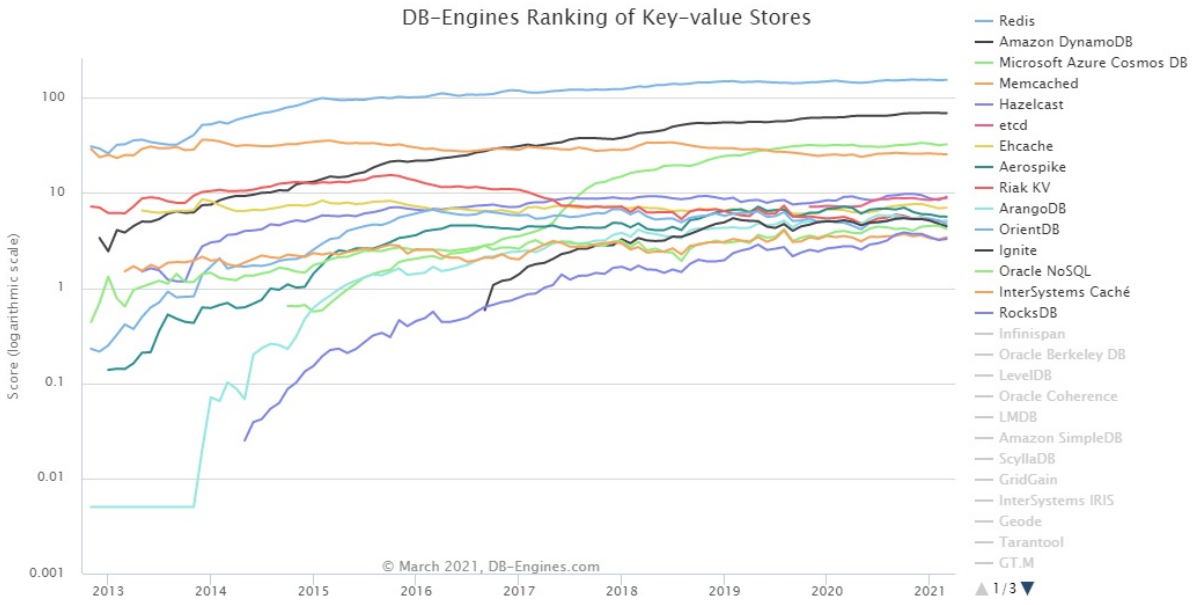
Nejčastěji využívaným přístupem je nastavování expirace. Hojně využívaná je rovněž možnost sestavení cache z nejběžnějších dotazů, případně doplněná o využití paralelních strojů. Samotné smazání obsahu cache po jakékoli aktualizaci v datech nemá příliš význam a existuje málo případů užití, pro které by tento přístup byl vhodný. Co se týče smazání pouze relevantních záznamů, tento postup je spíše teoretický a pro svou komplexitu také není příliš využíván.

Kapitola 6

Technologie využívané pro cachování

Současné technologie využívané pro cachování můžeme dělit na dvě kategorie. První jsou technologie, které se instalují a jsou dostupné lokálně na serveru. Typickými zástupci jsou například **Redis** a **Memcached**. Druhou kategorií jsou cloudová uložiště využívaná jako cache. Jedná se o poskytovatele těchto služeb, kteří nabízí cloudová úložiště pro cachování založená nejčastěji opět na technologiích Redis nebo Memcached.

Tato kapitola využívá mimo jiné také údaje ze serveru DB-Engines [1]. Na obrázku 6.1 je vidět graf porovnávající popularitu úložišť typu klíč-hodnota v letech 2013 až 2018. Z tohoto srovnání je patrné, že se po celou tuto dobu na pozici nejpopulárnějšího úložiště drží systém Redis. Proto byl tento systém posléze zvolen také pro implementaci praktické části této práce.



Obrázek 6.1: Srovnání popularity úložišť typu klíč-hodnota (zdroj: https://db-engines.com/)

6.1 Redis

Redis je paměťové datové úložiště, které se používá zejména jako cache, ale vzhledem k podpoře datové perzistence jej lze využívat také jako databázi. Podporuje širokou škálu datových typů i struktur. Data ukládá ve standardní podobě klíč-hodnota a nevyužívá SQL (jedná se tedy o systém z rodiny *NoSQL*). Kromě efektivity je oblíbený především pro svou jednoduchost. Redis umožňuje pouze běh na jediném vláknu (je *single-threaded*). Redis je open-source, jeho využívání je tedy zdarma.

Redis nabízí několik možností pro volbu politiky uvolňování záznamů (tzv. *data eviction policy*) [29]:

- *noeviction* – nedochází k automatickému mazání záznamů. Při překročení paměťového limitu vrací error.
- *allkeys-lru* – odstraňuje nejstarší klíče (z hlediska posledního využití).
- *volatile-lru* – odstraňuje nejstarší klíče jako *allkeys-lru*, ale pouze ty, u kterých je nastavena expirace.
- *allkeys-random* – odstraňuje klíče náhodně.
- *volatile-random* – odstraňuje klíče náhodně, ale pouze ty, u kterých je nastavena expirace.
- *volatile-ttl* – odstraňuje klíče s nejkratší zbývající životností (TTL) u kterých je nastavena expirace.
- *allkeys-lfu* – odstraňuje klíče na základě jejich odhadované frekvence dotazování (odstraňuje klíče s nejnižší frekvencí)
- *volatile-lfu* – odstraňuje klíče, u kterých je nastavena expirace, na základě jejich odhadované frekvence dotazování.

Pro zajištění trvalosti dat lze v pravidelných intervalech vytvářet snapshot systému nebo využít logování [30].

K dispozici je také placená verze Redis Enterprise, která nabízí zejména rozšířené možnosti škálování, dostupnosti, zálohování a ochrany proti chybám, zvýšená bezpečnostní opatření a online podporu [31].

Redis podporuje také spouštění skriptů v jazyce Lua. Toho lze využít pro zvýšení výkonu - namísto provádění složitějších manipulací s daty v mnoha oddělených krocích v rámci aplikaci lze jednoduše vše provést v rámci jediného příkazu obsahujícího Lua skript. Díky tomu nemusí docházet k opakované výměně dat mezi aplikací a Redisem a případným prodlevám.

Název	Redis
Primární způsob ukládání dat	Klíč-hodnota
SQL	Ne
Skriptování	Lua
Koncept transakcí	Optimistický přístup (verzování)
Podpora souběhu	Ne (běh na jediném vláknu)
Podpora trvalosti dat	Ano
Licence	Open Source

Tabulka 6.1: Redis - základní vlastnosti

Tabulka 6.1 uvádí základní vlastnosti databáze Redis.

6.2 Memcached

Jedná se o distribuovaný cachovací systém s všeobecným využitím, avšak zamýšlený především pro zrychlení dynamických webových aplikací minimalizací počtu databázových přístupů [32]. Memcached je škálovatelný systém, který umožňuje snadné přidávání nových uzlů, ale také zvyšování výpočetní kapacity díky paralelizaci (na rozdíl od systému Redis).

Memcached nepodporuje skriptování ani trvalost dat. Replikační metody jsou dostupné pouze jako součást *Repcached*, což je patch pro Memcached poskytující tuto funkcionalitu [33]. Na rozdíl od technologie Redis podporuje pouze jediný datový typ, kterým je textový řetězec (*string*). Jediný podporovaný způsob uvolňování cache je LRU.

Název	Memcached
Primární způsob ukládání dat	Klíč-hodnota
SQL	Ne
Skriptování	Ne
Koncept transakcí	Žádný
Podpora souběhu	Ano
Podpora trvalosti dat	Ne
Licence	Open Source

Tabulka 6.2: Memcached - základní vlastnosti

Tabulka 6.2 uvádí základní vlastnosti databáze Memcached.

6.3 Hazelcast

Hazelcast je systém od stejnojmenné společnosti, který nabízí vytváření paměťových struktur (*data grid*) se všestranným využitím. Často se využívá právě jako distribuovaná cache. Může běžet buď jako nainstalovaný software na serveru (*on-premises*), nebo v cloudu, virtuálně nebo v Docker kontejnerech. Hazelcast podporuje replikaci a zajišťuje také trvalost dat.

Jedná se o více vláknový systém založený na jazyku Java, proto se také často používá v Java aplikacích. Hazelcast však poskytuje klientskou podporu i ve většině hlavních programovacích jazycích [23].

Název	Hazelcast
Primární způsob ukládání dat	Klíč-hodnota
SQL	Ano
Skriptování	Ano
Koncept transakcí	Jedno nebo dvou fázové potvrzování, opakovatelné čtení, izolace read-committed
Podpora souběhu	Ano
Podpora trvalosti dat	Ano
Licence	Open Source

Tabulka 6.3: Hazelcast - základní vlastnosti

Tabulka 6.3 uvádí základní vlastnosti databáze Hazelcast.

6.4 Aerospike

Aerospike je hybridní systém, který dokáže ukládat hodnoty jak do paměti RAM, tak na SSD disk [23]. Využívá tím tak stále se zvyšujících propustností těchto komponent. Kromě standardních řetězců podporuje také ukládání dalších datových typů, jako jsou seznamy, mapy nebo serializované objekty. Jedná se o více vláknový systém, takže může těžit z více jádrových procesorů a poskytovat tak vysoký výkon.

Název	Aerospike
Primární způsob ukládání dat	Klíč-hodnota
SQL	AQL – Aerospike Query Language
Skriptování	Lua
Koncept transakcí	Atomické spouštění operací
Podpora souběhu	Ano
Podpora trvalosti dat	Ano
Licence	Open Source

Tabulka 6.4: Aerospike - základní vlastnosti

Tabulka 6.4 uvádí základní vlastnosti databáze Aerospike.

6.5 NCache

NCache je škálovatelná distribuovaná cache určená pro .NET / .NET Core. Podporuje replikaci dat s volitelným stupněm konzistence i trvalost dat. Je k dispozici také Enterprise verze, která poskytuje více možností data-partitioning, větší výkon WAN replikace a mnoho dalších funkcí.

Název	NCache
Primární způsob ukládání dat	Klíč-hodnota
SQL	QQL
Skriptování	Ne
Koncept transakcí	Optimistický přístup (verzování)
Podpora souběhu	Ano
Podpora trvalosti dat	Ano
Licence	Open Source

Tabulka 6.5: NCache - základní vlastnosti

Tabulka 6.5 uvádí základní vlastnosti databáze NCache.

6.6 Ehcache

Ehcache je cache pro Java aplikace. Je robustní a integrovatelná s populárními knihovnami a frameworky [34].

Název	Ehcache
Primární způsob ukládání dat	Klíč-hodnota
SQL	Ne
Skriptování	Ne
Koncept transakcí	Podpora Java Transaction API (JTA)
Podpora souběhu	Ano
Podpora trvalosti dat	Ano
Licence	Open Source

Tabulka 6.6: Ehcache - základní vlastnosti

Tabulka 6.6 uvádí základní vlastnosti databáze Ehcache.

6.7 Cloudové služby

V dnešní době jsou také velice rozšířené cloudové alternativy, které eliminují nutnost využívat pro cache vlastní paměťové zdroje. Od výše uvedených technologií se liší tím, že data nejsou uložena v paměti na serverech aplikací (a případně trvale na lokálním disku), ale v cloudu na straně poskytovatele této služby. Výhodou využívání cloudových úložišť může být především zakoupení konkrétní požadované kapacity paměti a kvality služeb a možnost tyto parametry v budoucnu upravovat podle potřeby.

Jedním ze zástupců je například Amazon ElastiCache, což je in-memory datové úložiště a cachovací služba od společnosti Amazon Web Services. Jedná se o cloudové úložiště, které podporuje Memcached a Redis (*ElastiCache for Redis*) [35]. Tabulka 6.7 ukazuje cenu za jednu standardní instanci pro Redis o kapacitě 60,78 GiB a 16 vCPU.

Název služby	**Cena za hodinu**	**Cena za měsíc**
Amazon ElastiCache for Redis	$1,482	$1081,9

Tabulka 6.7: Cena služby Amazon ElastiCache pro modelový příklad ke dni 10.2.2021 (vypočteno z https://calculator.aws/#/createCalculator/ElastiCache)

Dalším příkladem je Azure Redis Cache (také Azure for Redis). Jedná se o službu od společnosti Microsoft, která je založena na technologii Redis a opět poskytuje cloudové úložiště pro cache [36].

Posledním příkladem je Memorystore for Redis od společnosti Google. Opět se jedná o cloudové úložiště založené na technologii Redis. Tabulka 6.8 ukazuje cenu za základní instanci o kapacitě 64 GiB.

Název služby	**Cena za hodinu**	**Cena za měsíc**
Google Cloud - Memorystore for Redis	$1,472	$1059,8

Tabulka 6.8: Cena služby Memorystore for Redis pro modelový příklad ke dni 10.2.2021 (vypočteno z https://cloud.google.com/products/calculator)

Kapitola 7

Implementace

V rámci praktické části této práce byly implementovány některé dříve analyzované postupy. Pro testování byla použita existující databáze - produktový katalog Gloffer v systému MySQL, čítající téměř 60 milionů záznamů. Tato data byla importována jak do vlastní testovací MySQL databáze, tak do dokumentově orientovaných systémů řízení báze dat MongoDB a Elasticsearch. Pro testování byl využit stroj s operačním systémem Ubuntu 20.04, 16 GB paměti RAM, procesorem s 8 jádry a pevným diskem o kapacitě 1 TB.

Pro každou z těchto tří databází byly připraveny tři druhy sad dotazů, kde každá sada obsahovala 1000 dotazů. Cílem bylo otestovat, jestli si dokáží vybrané databázové systémy poradit lépe s běžnými predikáty v dotazech, s fulltextovými dotazy nebo s kombinací obojího.

Prvním druhem byly dotazy s predikáty obsahujícími pouze konkrétní hodnoty atributů, tedy bez fulltextového vyhledávání (dále *parametrické dotazy*). Dalším druhem byly dotazy kombinující jak predikáty z předchozích parametrických dotazů, tak fulltextové vyhledávání (dále *kombinované dotazy*). Posledním druhem byly čistě fulltextové dotazy bez dalších specifikujících podmínek. Pro všechny tři typy byly vygenerovány následující sady dotazů:

1. unikátní dotazy
2. opakující se dotazy (50 % totožných dotazů, jediný dotaz se opakuje 500×)
3. opakující se dotazy (50 % totožných dotazů, každý dotaz se opakuje 2×)

Všechny skupiny dotazů byly pro porovnání časů spouštěny jak s implementovaným cachováním, tak bez něj.

Jak již bylo řečeno, byly připraveny dvě různé sady opakujících se dotazů, u kterých se lišil způsob opakování. Důvodem byla snaha otestovat, zda dochází ke zhoršení výkonu při větším počtu různých dotazů, které se opakují; když tedy dochází více k obměně stránek uložených v cache buffer. V první skupině jsou tedy dotazy, ve kterých se jediný dotaz opakuje 500× po vzoru *ABCBDB*. Tato skupina bude dále označována jako **rep500**. Druhá skupina obsahuje 500 unikátních dotazů, z nichž

se každý zopakuje 2×, vzorem *ABCABC*. Tato skupina bude označená **rep2**. Skupina unikátních dotazů bude značena jako **unique**.

Pro testování byla připravena Java aplikace, která se připojí k vybrané databázi a spustí postupně všechny várky dotazů, přičemž měří čas pro výsledné porovnání.

7.1 Cachování a invalidace

Ve zmíněné Java aplikaci je také implementováno cachování za použití technologie Redis, které funguje na principu popsaném na obrázku 3.1. Instalaci Redisu lze provést pomocí nástroje package manager, nicméně tento postup se nedoporučuje, protože poslední dostupná verze nemusí být vždy ta nejnovější. Místo toho se doporučuje stáhnout a zkompilovat přímo zdrojový kód [37]. Prvním krokem je stažení a rozbalení archivu z oficiálních webových stránek, což se provádí příkazy:

```
$ wget http://download.redis.io/redis-stable.tar.gz
$ tar xvzf redis-stable.tar.gz
```

Následuje přesunutí do příslušného adresáře a kompilace pomocí nástroje `make`:

```
$ cd redis-stable
$ make
```

Po úspěšné kompilaci lze v podadresáři `src` nalézt několik spustitelných souborů, z nichž nejpodstatnější jsou `redis-server` a `redis-cli`. Tyto spustitelné soubory je možné spouštět přímo z tohoto umístění, nebo je lze přesunout do vhodných adresářů, typicky pomocí příkazů:

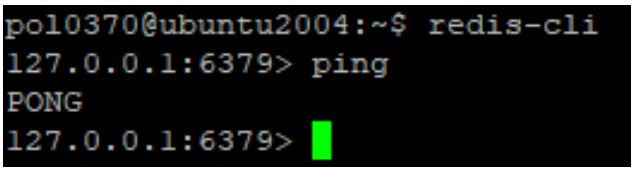
```
$ sudo cp src/redis-server /usr/local/bin/
$ sudo cp src/redis-cli /usr/local/bin/
```

Následně je již možné spustit Redis server:

```
$ redis-server
```

V případě, že je potřeba nepoužívat výchozí hodnoty a specifikovat konfigurační soubor, uvádí se cesta k tomuto souboru jako argument za tímto příkazem.

Pro ověření, zda je Redis spuštěn a funguje správně, lze využít `redis-cli`, který poskytuje rozhraní příkazového řádku, skrze který je možné testovat a používat Redis. Na obrázku 7.1 je vidět správná odpověď na příkaz `PING`, která signalizuje, že Redis běží a je funkční.



Obrázek 7.1: Ověření funkčnosti spuštěného serveru Redis

Pro implementaci v Java aplikaci byla využita knihovna `jedis`. Zdrojový kód této implementace je uveden ve výpisu 7.1. Pro uložení výsledků vyhledávání byl použit `HSET`, kde klíčem je SHA hash daného dotazu. Hodnotami jsou `dataHash` pro hash vráceného výsledku, který lze následně získat z cache pomocí metody `queryCacheForResult`, díky čemuž není nutné stejné výsledky vyhledávání ukládat v cache vícekrát. Dále `query` pro samotný dotaz pro případnou pozdější rekonstrukci a `category` jako zařazení produktu do kategorie pro odstranění neplatných dat celé této kategorie, viz kapitola 5.5.

```
// Získání uloženého výsledku daného dotazu z cache
// Pokud se tato data v cache nenacházejí, je vrácena hodnota null.
public String queryCache(String query) {
    String shaHash = getSha(query);
    String key = "query:" + shaHash;
    Map<String,String> result = jedis.hgetAll(key);
    countUp(shaHash);
    if (!result.isEmpty()) {
        String dataHash = result.get("dataHash");
        return queryCacheForResult("result:" + dataHash);
    } else return null;
}

// Uložení dotazu, výsledku vyhledávání a kategorie produktu do cache.
// Klíčem je SHA hash dotazu.
public void cache(String query, String result, String category) {
    String shaQueryHash = getSha(query);
    String shaResultHash = getSha(result);
    String queryKey = "query:" + shaQueryHash;
    String resultKey = "result:" + shaResultHash;
    String categoryKey = category.equals("") ? ("category:" + NO_CATEGORY) : (
        "category:" + category);
    cacheResult(resultKey, result);
    Map<String, String> values = new HashMap<String, String>();
```

```
        values.put("dataHash", shaResultHash);
        values.put("query", query);
        values.put("category", categoryKey);
        jedis.hset(queryKey, values);
        jedis.expire(queryKey, TTL);
        jedis.lpush(categoryKey, queryKey);
        jedis.lpush(categoryKey, resultKey);
    }

    // Uložení výsledku vyhledávání do cache, klíčem je jeho hash
    public void cacheResult(String resultKey, String result) {
        String foundResult = queryCacheForResult(resultKey);
        if (foundResult == null) {
            jedis.set(resultKey, result);
            jedis.expire(resultKey, TTL);
        }
    }

    // Získání výsledku z jeho hashe (deduplikace výsledků vyhledávání)
    public String queryCacheForResult(String resultKey) {
        String result = jedis.get(resultKey);
        if (!(result == null) && !result.equals("")) return result;
        else return null;
    }

    // Navýšení hodnoty počítadla pro dotaz na daný klíč
    private void countUp(String key) {
        Double currentCount = jedis.zscore(COUNTS_SET, key);
        if (currentCount == null) {
            jedis.zadd(COUNTS_SET, 1, key);
        } else {
            jedis.zadd(COUNTS_SET, ++currentCount, key);
        }
    }
```

Výpis 7.1: Cachování výsledků vyhledávání v Redis

Ve výpisu 7.1 je možné vidět, že při ukládání klíče do cache se zároveň vloží tento klíč do seznamu klíčů spadající pod danou kategorii pomocí příkazu `jedis.lpush`. Díky tomu je možné v případě

potřeby identifikovat a smazat pouze klíče, které byly postiženy změnou dat týkajících se určité kategorie produktů. Ve výpisu je uvedena také metoda `countUp`, která navýší hodnotu počítadla v kolekci s názvem `countset`, která byla v úložišti Redis vytvořena za tímto účelem. Díky tomu lze před kompletním smazání celého obsahu cache identifikovat nejčastěji pokládané dotazy a následně je znovu položit databázi a obnovit tak automaticky výsledky v cache. Ve výpisu 7.2 je uvedena ukázka v Redis CLI pro nalezení 5 dotazů s nejvyšším skóre (nejvyšší hodnotou počítadla) pomocí příkazu `ZREVRANGEBYSCORE`, který vrací prvky od nejvyšší hodnoty skóre po nejnižší. Následně je pomocí příkazu `HGET` vráceno pole `query` obsahující plné znění zvoleného dotazu.

```
> ZREVRANGEBYSCORE countset +inf -inf LIMIT 0 5
 1) "query:50C8FD2A26FFCA0A7798C27703978D4E6B11062D1F820C6F678EB13EA5D3CBB9"
 2) "query:B594CA3C303EC2591FB08448BF604E634AA9DC173E38A6DEB65E9338586F8CFA"
 3) "query:0A60A561FAD96F9C4DE04ECB33FCF0314778419275A2F8F49C1A6B49BA7904CA"
 4) "query:C6ED9DD46EE3C0FBBEB9E10EEB1F317A59729384DED1A2B1B6DD4DE0D330AD89"
 5) "query:9F73B0B783E2339194715959154AEE20EE60FE0036E23A373C9E954740FAF273"
> HGET "query:50C8FD2A26FFCA0A7798C27703978D4E6B11062D1F820C6F678EB13EA5D3CBB9"
    query
"{\"property.multi.value_title\": \"Tripod\", \"pricing.price\" : {$gte: 150, $lte
    : 200}}"
```

Výpis 7.2: Nalezení dotazů s nejvyšší četností dotazování v programu Redis CLI

V aplikaci byla použita metoda TTL (nastavování expirace) pomocí metody `jedis.expire`. Byly však také naimplementovány metody pro smazání a znovusestavení cache po změně v datech.

```
    // Vrátí prvních N klíčů s nejvyšší hodnotou počítadla
    public Set<String> getTopKeys(int n) {
        return jedis.zrevrangeByScore(COUNTS_SET, Double.POSITIVE_INFINITY, Double
            .NEGATIVE_INFINITY, 0, n);
    }

    // Vrátí všechny klíče s hodnotou počítadla vyšší než N
    public Set<String> getKeysAbove(int n) {
        return jedis.zrevrangeByScore(COUNTS_SET, Double.POSITIVE_INFINITY, n);
    }

    // Vrátí všechny klíče spadající pod danou kategorii
    public List<String> getKeysByCategory(String category, boolean
        includeKeysWithoutCategory) {
        List<String> result = new ArrayList<>(jedis.lrange("category:" + category,
            0, -1));
```

```
        if (includeKeysWithoutCategory && !category.equals(NO_CATEGORY)) {
            result.addAll(jedis.lrange("category:" + NO_CATEGORY, 0, -1));
        }
        return result;
    }

    // Odstraní všechny klíče spadající pod danou kategorii
    public void flushByCategory(String category, boolean
        includeKeysWithoutCategory) {
        category = "category:" + category;
        String[] arguments = {category};
        String script = "local keys = redis.call('lrange', ARGV[1], 0, -1) for i
            =1,#keys,5000 do~redis.call('del', unpack(keys, i, math.min(i+4999, #
            keys))) end";
        jedis.eval(script, 0, arguments);

        if (includeKeysWithoutCategory && !category.equals(NO_CATEGORY)) {
            category = "category:" + NO_CATEGORY;
            String[] noCategoryArguments = {category};
            script = "local keys = redis.call('lrange', ARGV[1], 0, -1) for i=1,#
                keys,5000 do~redis.call('del', unpack(keys, i, math.min(i+4999, #
                keys))) end";
            jedis.eval(script, 0, noCategoryArguments);
        }
    }
```

Výpis 7.3: Metody pro znovu sestavení cache v třídě Cacher

Ve výpisu 7.3 jsou tyto metody uvedeny. Metody `getTopKeys` a `getKeysAbove` slouží k získání klíčů s nejvyšší hodnotou počítadla ze souboru `countset`, z nichž lze pak pomocí metody `getQueryByKey` získat znění konkrétního dotazu. Pokud je potřeba znovu sestavit cache pro konkrétní kategorii, je možné použití metody `getKeysByCategory`, která vrátí veškeré klíče související s touto kategorií a případně také klíče bez nastavené kategorie. Odstranění aktuálně uložených klíčů z dané kategorie a jejich hodnot se provádí v metodě `flushByCategory`, která smaže pouze klíče uložené v poli pro danou kategorii, případně také klíče bez nastavené kategorie. Implementace této metody je provedena za použití Lua skriptu a příkazu `jedis.eval`.

```
    // Znovu sestaví cache s využitím n klíčů s nejvyšší hodnotou počítadla
    public void rebuildCacheTopN(int n) {
        List<String> keysList = new ArrayList<>(cacher.getTopKeys(n));
```

```
        List<String> queries = new ArrayList<>();
        for (String key : keysList) {
            if (key.startsWith("query:")) {
                String query = cacher.getQueryByKey(key);
                queries.add(query);
            }
        }
        cacher.flushAll();
        rebuildQueries(queries);
    }

    // Odstraní klíče spadající pod danou kategorii a znovu sestaví cache
    public void rebuildCacheByCategory(String category, boolean
        includeKeysWithoutCategory) {
        List<String> keysList = cacher.getKeysByCategory(category,
            includeKeysWithoutCategory);
        List<String> queries = new ArrayList<>();
        for (String key : keysList) {
            if (key.startsWith("query:")) {
                String query = cacher.getQueryByKey(key);
                queries.add(query);
            }
        }
        cacher.flushByCategory(category, includeKeysWithoutCategory);
        rebuildQueries(queries);
    }
```

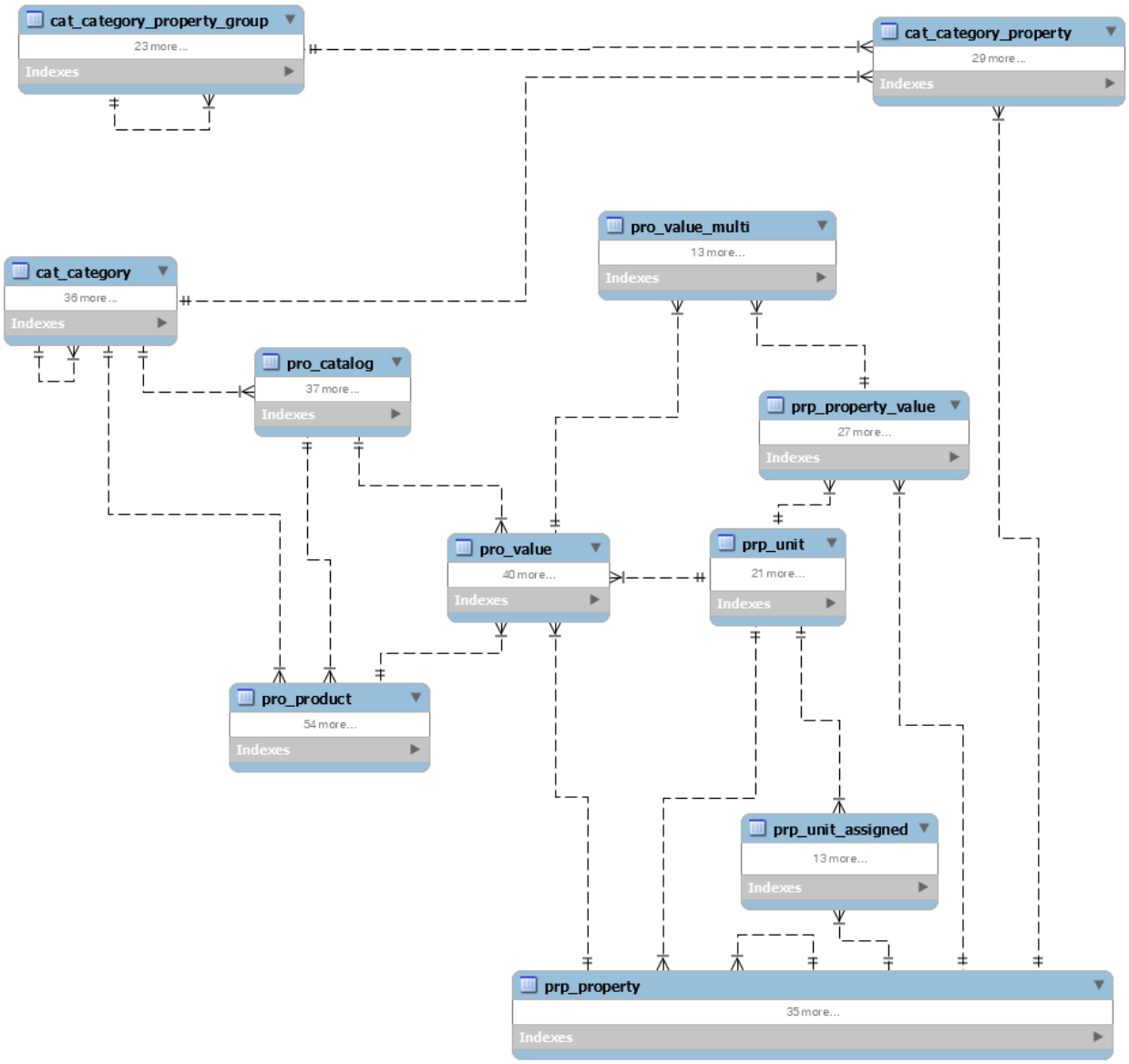
Výpis 7.4: Využití metod pro znovu sestavení cache

Výpis 7.4 zobrazuje konkrétní využití metod pro obnovení cache.

Následující kapitoly popisují výsledky měření provádění výše uvedených sad dotazů. Všechny časy uvedené v jednotlivých grafech jsou v sekundách a představují čas, který trvalo vykonání celé sady 1000 dotazů těsně po sobě. Měření bylo vždy provedeno třikrát a uvedené časy představují výsledný průměr.

7.2 MySQL

Jak již bylo uvedeno, byla použita data z originální databáze Gloffer, konkrétně její produktový katalog. Model této databáze ukazuje obrázek 7.2.



Obrázek 7.2: Model produktového katalogu Gloffer v databázi MySQL.

Tato databáze využívá generický databázový model. Vlastnosti jednotlivých záznamů nejsou v tomto případě uloženy jako hodnoty předem definovaných sloupců. Namísto toho jsou v samostatné tabulce uloženy názvy všech existujících atributů. Ve vazební tabulce je pak pro každou vlastnost (atribut) daného produktu jeden záznam, který obsahuje ID produktu, ID atributu a hodnotu tohoto atributu (jako konkrétní hodnotu nebo jako ID jedné z hodnot, které jsou také uloženy v samostatné tabulce).

Tento model je zvlášť výhodný právě pro produktové katalogy, protože pro každý produkt existuje celá řada možných atributů a výsledná tabulka by měla příliš mnoho sloupců. Zároveň

tento model poskytuje vyšší flexibilitu z hlediska úpravy databázového schématu, protože v případě nutnosti přidání nového atributu není potřeba provádět nákladnou operaci změny schématu celé databázové tabulky. Namísto toho lze dynamicky přidávat nové vlastnosti pouhým vložením nového záznamu.

Nevýhodou je však dotazování, jelikož je příprava dotazů nad tímto modelem velmi komplexní a musí docházet ke složitým dotazovým konstrukcím a spojování mnoha tabulek. S ohledem na tyto komplikace je dobře viditelná výhoda dokumentově orientovaných databází, které nabízí flexibilní datové schéma.

Data z původní databáze Gloffer byla naimportována do lokální databáze MySQL, jejíž instalace byla provedena v následujících krocích. Nejprve byl nainstalován balíček `mysql-server`:

```
$ sudo apt update
$ sudo apt install mysql-server
```

Následně byl spuštěn skript `mysql_secure_installation`, který uživatele provede sérií bezpečnostních nastavení:

```
$ sudo mysql_secure_installation
```

Pro změnu konfigurace je možné upravit soubor `mysqld.cnf` pomocí příkazu:

```
$ sudo nano /etc/mysql/mysql.conf.d/mysqld.cnf
```

V tomto souboru lze nakonfigurovat různé adresáře a další proměnné, ale také velikost cache bufferu. Ta byla v tomto případě nastavena na 7 GB:

```
innodb_buffer_pool_size = 7G
```

Následné spuštění služby MySQL se provádí příkazem:

```
$ sudo systemctl start mysql.service
```

Pro ověření, zda služba běží, je možné použít příkaz:

```
$ service mysql status
```

Tento příkaz ověří status této služby a zobrazí ho na výstupu, jako lze vidět na obrázku 7.3.

```
pol0370@ubuntu2004:/$ service mysql status
● mysql.service - MySQL Community Server
     Loaded: loaded (/lib/systemd/system/mysql.service; enabled; vendor preset: enabled)
     Active: active (running) since Sat 2021-02-27 19:27:55 CET; 1min 16s ago
    Process: 190533 ExecStartPre=/usr/share/mysql/mysql-systemd-start pre (code=exited, status=0/SUCCESS)
   Main PID: 190551 (mysqld)
     Status: "Server is operational"
      Tasks: 41 (limit: 19075)
     Memory: 1.2G
     CGroup: /system.slice/mysql.service
             └─190551 /usr/sbin/mysqld
```

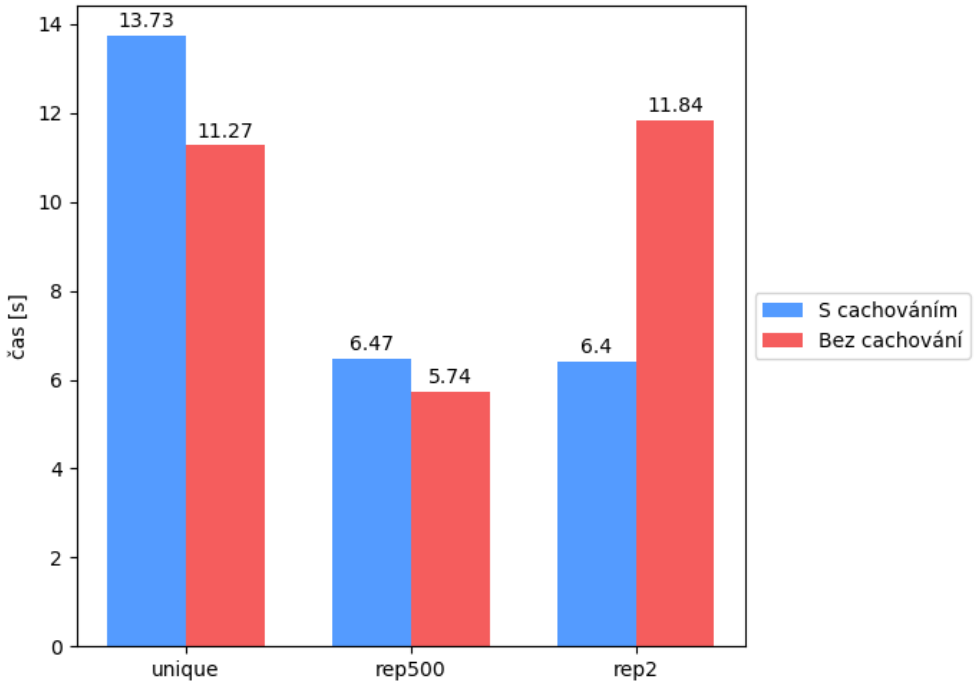
Obrázek 7.3: Status spuštěné služby MySQL

Po nainstalování byla do databáze naimportována data. Pro fulltextové vyhledávání byl navíc nad databází vytvořen index příkazem uvedeným ve výpisu 7.5.

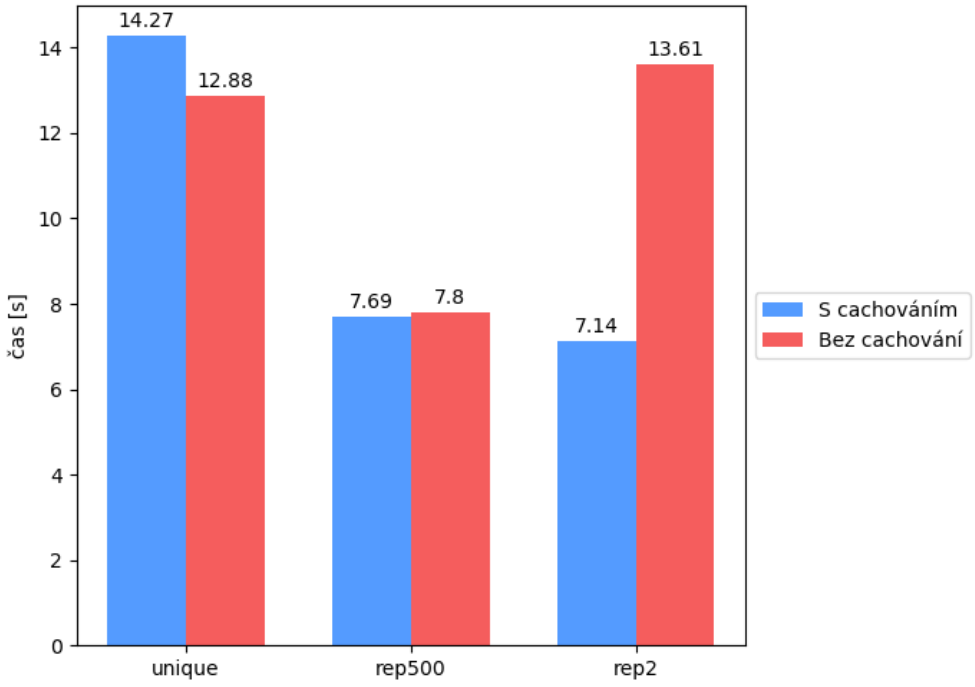
```
CREATE FULLTEXT INDEX IX_pro_catalog_fulltext
ON pro_catalog(pro_catalog_content)
```

Výpis 7.5: Vytvoření fulltextového indexu v databázi MySQL

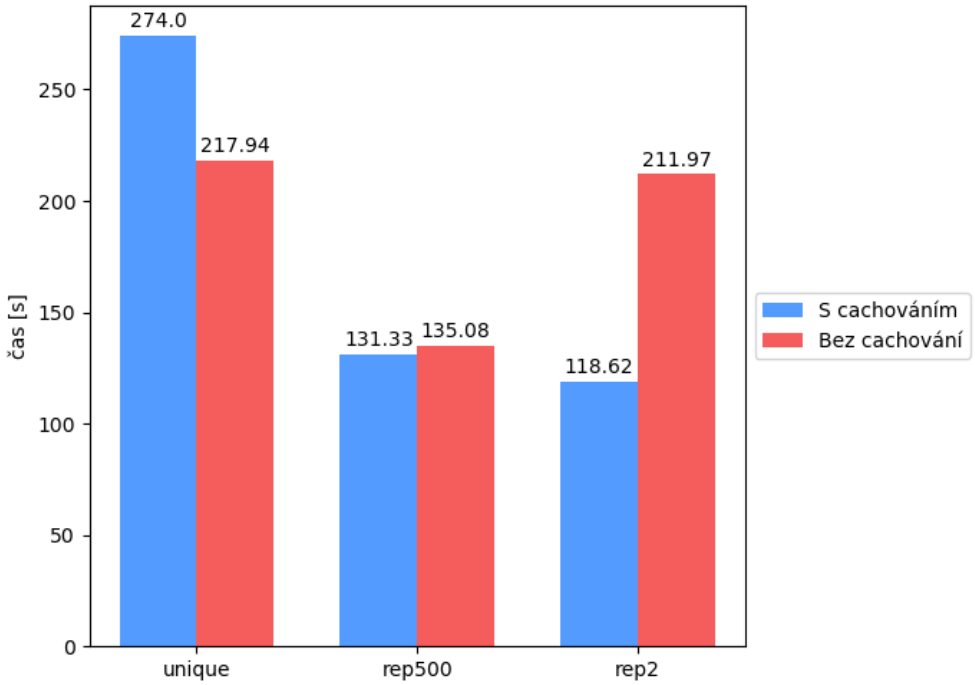
Výsledky měření ukazují grafy na obrázku 7.4. Z grafu pro parametrické dotazy je patrné, že jak u skupiny **unique**, tak u skupiny **rep500** nedošlo k žádné výrazné změně při spuštění dotazů s cachováním oproti spuštění bez něj. Ovšem u skupiny **rep2** již došlo k výraznému rozdílu: bez cachování se čas zvýšil o zhruba 85 %. To je dáno nutností vyřadit z paměti cache buffer některé stránky, které již delší dobu nebyly použity, aby bylo možné do něj uložit aktuálně použité stránky. Při příštím spuštění stejného dotazu tak už požadované stránky nemusely být uloženy v cache buffer, což by znamenalo, že bylo potřeba je znovu načíst z disku. Tento problém byl však eliminován při použití cachování v technologii Redis, kde byl čas téměř totožný pro obě skupiny opakujících se dotazů.



(a) Parametrické dotazy



(b) Kombinované dotazy



(c) Fulltextové dotazy

Obrázek 7.4: Výsledky měření v databázi MySQL

Výsledné časy byly oproti parametrického vyhledávání nepatrně vyšší u kombinovaných dotazů, protože bylo nutné z výsledků parametrického vyhledávání následně ještě vyfiltrovat ty, které neobsahují hledaný fulltextový výraz. Rozdíly však i v tomto případě vypovídají, že u skupiny **rep2** se bez cachování čas zvýšil o téměř 91 %.

Fulltextové vyhledávání již trvalo podstatně déle, ale poměrově vykazují výsledky stejný vzorec. U skupiny **rep2** se bez cachování čas zvýšil až o téměř 79 %.

7.3 MongoDB

Instalace databáze MongoDB je uvedena v následujících krocích. Nejprve byl importován veřejný klíč pomocí následujícího příkazu:

```
$ wget -qO - https://www.mongodb.org/static/pgp/server-4.4.asc | sudo apt-key add -
```

Definice repozitáře byla uložena do souboru `/etc/apt/sources.list.d/mongodb-org-4.4.list`:

```
$ echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/4.4
multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-4.4.list
```

Dále byl nainstalován balíček MongoDB:

```
$ sudo apt-get update
$ sudo apt-get install -y mongodb-org
```

Pro změnu konfigurace je možné upravit soubor `mongod.conf` pomocí příkazu:

```
$ sudo nano /etc/mongod.conf
```

V tomto souboru lze například změnit adresář pro data, logování a podobně. V tomto kroku byla nastavena také velikost WiredTiger cache na 7 GB:

```
storage:
   dbPath: /data/db
   wiredTiger:
      engineConfig:
         cacheSizeGB: 7
```

Následně je už možné spustit službu `mongod`:

```
$ sudo systemctl start mongod
```

Pro ověření, zda služba běží, je možné použít příkaz:

```
$ service mongod status
```

Tento příkaz ověří status této služby a zobrazí ho na výstupu, jako lze vidět na obrázku 7.5.

```
pol0370@ubuntu2004:~$ service mongod status
● mongod.service - MongoDB Database Server
     Loaded: loaded (/lib/systemd/system/mongod.service; enabled; vendor preset: enabled)
     Active: active (running) since Tue 2021-02-23 10:51:07 CET; 4 days ago
       Docs: https://docs.mongodb.org/manual
   Main PID: 49900 (mongod)
     Memory: 6.7G
     CGroup: /system.slice/mongod.service
             └─49900 /usr/bin/mongod --config /etc/mongod.conf
```

Obrázek 7.5: Status spuštěné služby MongoDB

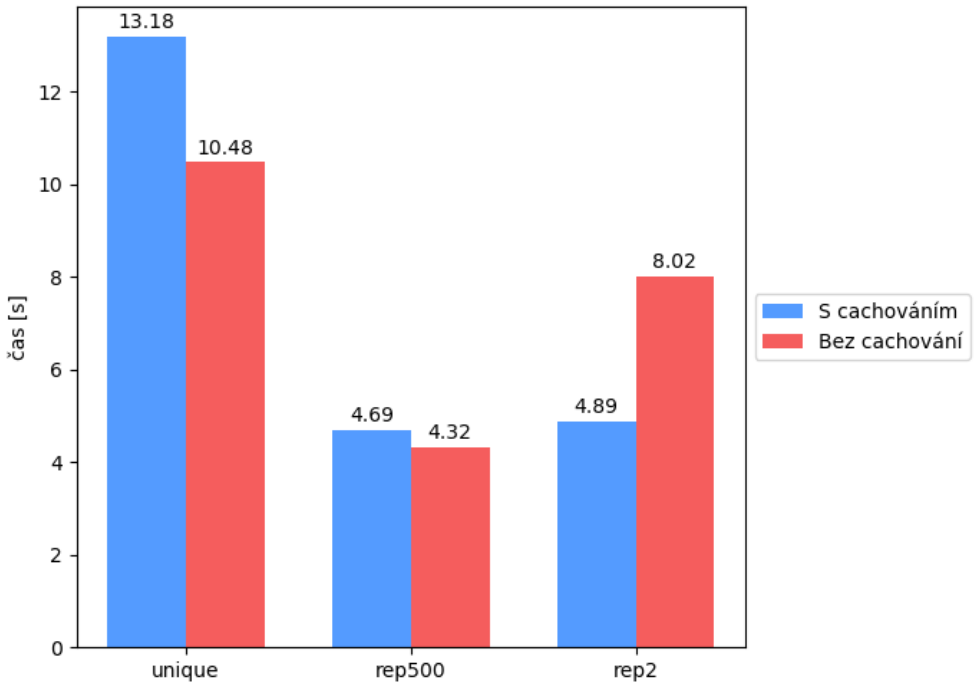
Pro naplnění databáze byla využita stejná data, ovšem pocházející z databáze gloffer_cache. Jedná se o databázi, která již také byla k dispozici v rámci původního databázového systému. Data

jsou ve formátu JSON a byla sestavena z relační databáze Gloffer. Tato data byla naimportována do vytvořené kolekce `gloffer_cache`.

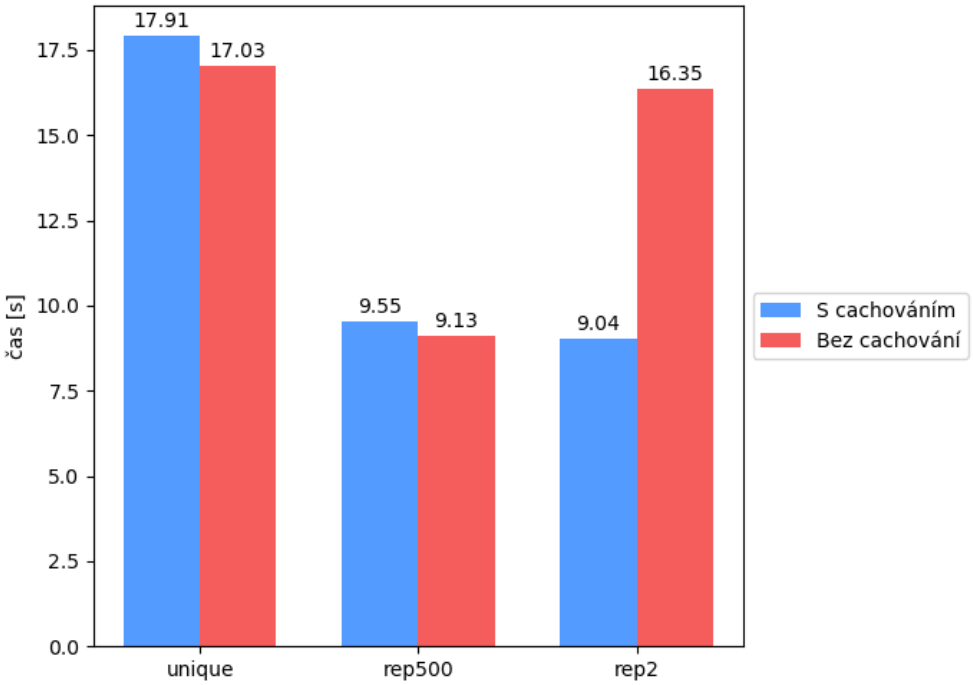
MongoDB nabízí možnost vytvoření textového indexu nad kolekcí. Pro specifikování jazyka vyhledávání je vhodné si vybrat při sestavování indexu jeden z 15 aktuálně nabízených jazyků. Čeština však ke dnešnímu dni není podporovaná, což negativně ovlivňuje fulltextové vyhledávání a jeho výsledky. Pro umožnění fulltextového vyhledávání byl vytvořen textový index bez specifikovaného jazyka, jak je ukázáno ve výpisu 7.6 [38].

```
db.gloffer_cache.createIndex( {"translate.cs.content": "text"}, {default_language:
    "none", language_override: "none"} )
```

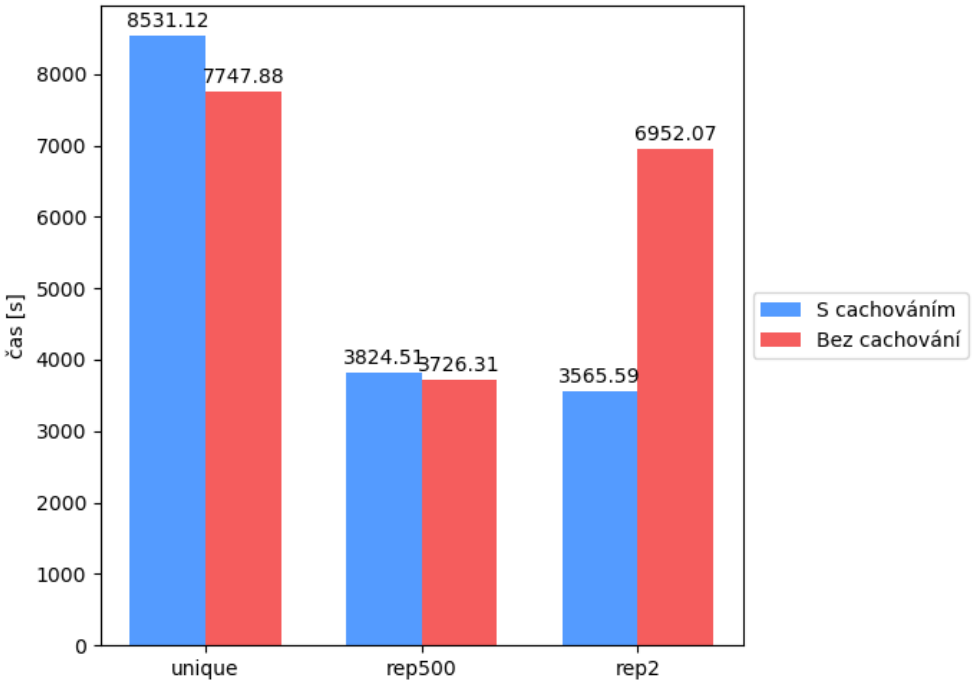
Výpis 7.6: Vytvoření textového indexu v databázi MongoDB



(a) Parametrické dotazy



(b) Kombinované dotazy



(c) Fulltextové dotazy

Obrázek 7.6: Výsledky měření v databázi MongoDB

Výsledky měření ukazují grafy na obrázku 7.6. Provádění parametrických dotazů i v tomto případě vykázalo rozdíl u skupiny **rep2**, kde se bez použití cachování čas zvýšil o 64 %. I v tomto případě byl tento rozdíl u všech měření eliminován použitím cache - u obou skupin opakujících se dotazů **rep500** i **rep2** bylo s cachováním dosaženo stejných výsledků.

Ještě výraznějšího zvýšení času u skupiny **rep2** bylo dosaženo u testování kombinovaných dotazů. Bez cachování se v tomto případě zvýšil čas až o 81 %. Všechny časy se také u kombinovaného vyhledávání oproti pouze parametrickému vyhledávání zvýšily zhruba na dvojnásobek, což je opět dáno nutností filtrovat výsledky, které byly nalezeny na základě parametrického vyhledávání.

Je důležité poznamenat, že samotné fulltextové vyhledávání, které využívalo vytvořeného textového indexu, trvalo výrazně déle - čas se oproti parametrickému vyhledávání zvýšil až o stonásobky. Z hlediska fulltextového vyhledávání tak databáze MongoDB neprokázala dobrý výkon. I zde pak došlo k výraznému zhoršení u skupiny **rep2** bez použití cachování, čas byl vyšší až o 95 %.

7.4 Elasticsearch

Instalace služby Elasticsearch byla provedena v následujících krocích. Nejprve byl stažen a nainstalován veřejný klíč:

```
$ wget -qO - https://artifacts.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add -
```

Definice repozitáře byla uložena do souboru `/etc/apt/sources.list.d/elastic-7.x.list`:

```
$ echo "deb https://artifacts.elastic.co/packages/7.x/apt stable main" | sudo tee -a
/etc/apt/sources.list.d/elastic-7.x.list
```

Dále byl nainstalován samotný balíček Elasticsearch Debian:

```
$ sudo apt-get update && sudo apt-get install elasticsearch
```

Pro konfiguraci služby je možné změnit hodnoty proměnných, například adresářů pro data a jiné soubory, v konfiguračním souboru `elasticsearch.yml`:

```
$ sudo nano /etc/elasticsearch/elasticsearch.yml
```

Pro ladění výkonu lze také upravit některá nastavení JVM, zejména velikost haldy (*heap size*). V dokumentaci je uvedeno, že Elasticsearch určuje velikost haldy automaticky a pro většinu produkčních prostředí je doporučeno tato nastavení neměnit. Pokud je však žádoucí tyto změny provést, lze toho dosáhnout úpravou souboru `jvm.options`. Velikost haldy určují proměnné `Xms` (minimální

velikost haldy) a `Xmx` (maximální velikost haldy), přičemž obě tyto proměnné musí být nastaveny na stejnou hodnotu. Zároveň by hodnoty těchto proměnných neměly přesáhnout 50 % celkové dostupné paměti. S vyšší velikostí haldy se zvyšuje množství paměti, kterou může Elasticsearch využít pro vnitřní cachování, ovšem také to snižuje paměť, kterou může používat operační systém pro cachování v souborovém systému a může dojít ke snížení výkonu vlivem garbage collection [39, 40].

Byla otestována jak varianta s výchozím nastavením, tak varianta s velikostí haldy 7 GB. Při výchozím nastavení byly všechny výsledné časy o něco vyšší (zvýšení se pohybovalo okolo zhruba 10 %), proto byla pro konečné testování zvolena varianta s upravenou velikostí haldy na 7 GB. Tuto úpravu lze provést následujícím příkazem:

```
$ sudo nano /etc/elasticsearch/elasticsearch.yml
-Xms7g
-Xmx7g
```

Po dokončení instalace a konfigurace je možné nastavit automatické spouštění služby Elasticsearch. Pro to je potřeba znovu načíst konfiguraci systemd a následně povolit službu Elasticsearch:

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable elasticsearch.service
```

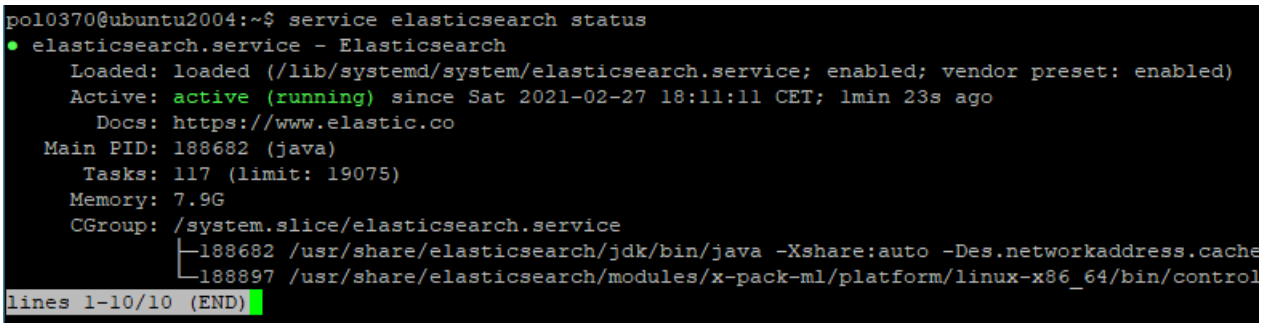
Samotné spuštění služby Elasticsearch se provádí příkazem:

```
$ sudo systemctl start elasticsearch.service
```

Pro ověření, zda služba běží, je možné použít příkaz:

```
$ service elasticsearch status
```

Tento příkaz ověří status této služby a zobrazí ho na výstupu, jako lze vidět na obrázku 7.7.



Obrázek 7.7: Status spuštěné služby Elasticsearch

Pro porovnání byly připraveny tři různé indexy, a to s jedním, pěti a deseti shardy. Jejich počet se určuje parametrem `"number_of_shards": "n"`. Vytvoření indexu obsahujícího jeden shard je uvedeno ve výpisu 7.7.

```
PUT produkty
{
    "settings": {
        "index": {
            "number_of_shards": "1",
            "analysis": {
                "filter": {
                    "remove_duplicities": {
                        "type": "unique",
                        "only_on_same_position": "true"
                    },
                    "delimiter": { "type": "word_delimiter" },
                    "ascii_folding": {
                        "type": "asciifolding",
                        "preserve_original": "false"
                    },
                    "hunspell_cs_CZ": {
                        "recursion_level": "0",
                        "locale": "cs_CZ",
                        "type": "hunspell",
                        "dedup": "true"
                    }
                },
                "analyzer": {
                    "cs_analyzer_html_strip": {
```

```
                    "filter": [
                        "hunspell_cs_CZ",
                        "lowercase",
                        "remove_duplicities",
                        "ascii_folding"
                    ],
                    "char_filter": [ "html_strip" ],
                    "type": "custom",
                    "tokenizer": "standard"
                }
            }
        }
    }
}
```

Výpis 7.7: Vytvoření indexu v databázi Elasticsearch

Součástí tohoto výpisu je také konfigurace analyzátorů pro tento index. Nebyl použit vestavěný český stemmer, namísto něj byl použit externí slovníkový stemmer `hunspell`, a to vzhledem k lepší relevanci vracených výsledků [9]. Dále byl použit tokenový filtr `remove_duplicities` pro odstranění duplicitních tokenů, `delimiter` na úrovni slov a tokenový filtr `ascii_folding` pro odstranění diakritiky a dalších unicode znaků do jejich ASCII ekvivalentů. V rámci analyzátoru byl rovněž použit znakový filtr `html_strip` pro odstranění prvků HTML.

Mapování zvolené pro index použitý pro testování je uvedeno ve výpisu 7.8. Pro některá pole typu `text` bylo nastaveno použití dříve definovaného analyzátoru.

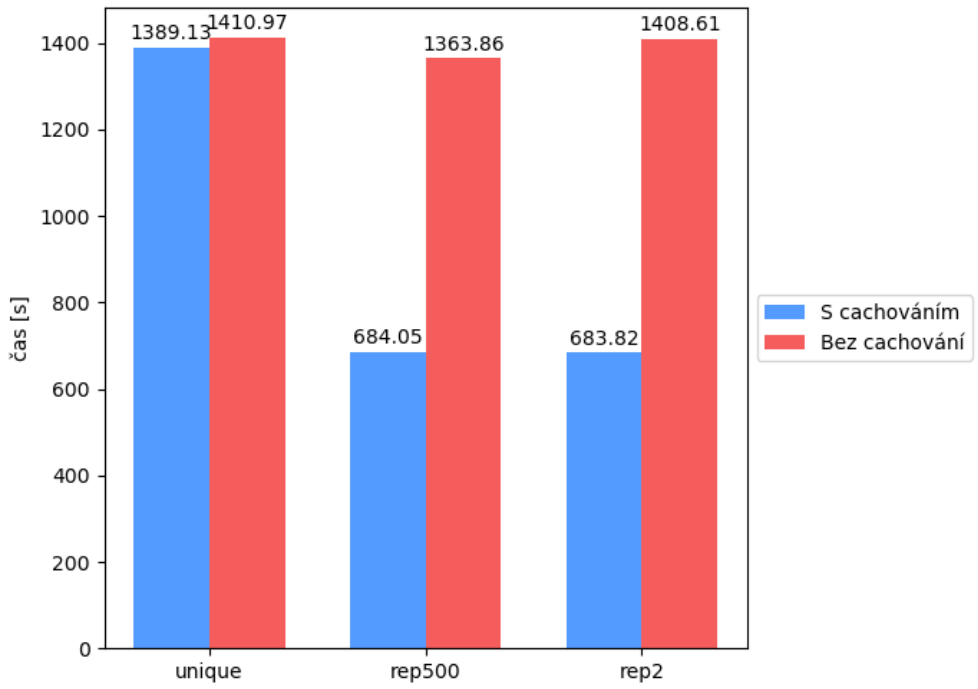
```
PUT /produkty/_mapping
{
    "properties": {
        "id": { "type": "integer" },
        "code": { "type": "integer" },
        "active": { "type": "keyword" },
        "country": { "type": "keyword" },
        "pricing": {
            "type": "nested",
            "properties": {
                "price": { "type": "double" },
                "currency": { "type": "keyword" }
            }
```

```
},
"category": {
    "type": "nested",
    "properties": {
        "id": { "type": "integer" },
        "code": { "type": "keyword" }
    }
},
"language": { "type": "keyword" },
"property": {
    "type": "nested",
    "properties": {
        "id": { "type": "integer" },
        "code": { "type": "keyword" },
        "slug": { "type": "keyword" },
        "type": { "type": "keyword" }
    }
},
"translate": {
    "type": "nested",
    "properties": {
        "cs": {
            "type": "nested",
            "properties": {
                "slug": { "type": "text" },
                "label": { "type": "text" },
                "title": { "type": "text" },
                "content": {
                    "type": "text",
                    "analyzer": "cs_analyzer_html_strip"
                },
                "description": {
                    "type": "text",
                    "analyzer": "cs_analyzer_html_strip"
                },
                "category_path": { "type": "text" }
            }
        }
```

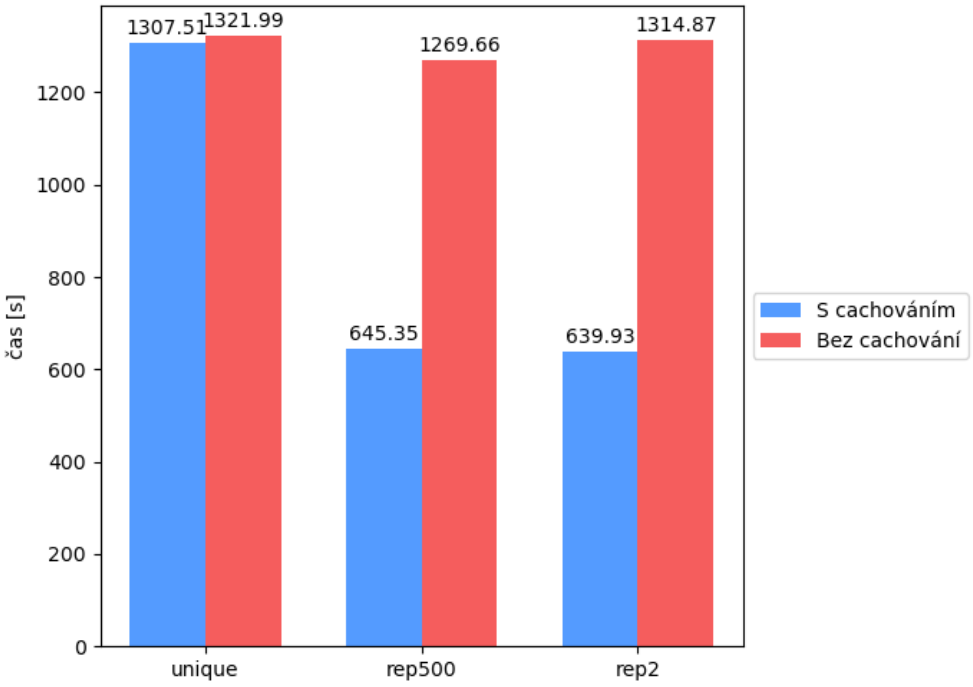
```
            }
        },
        "visibility": { "type": "keyword" },
        "category_id": { "type": "integer"
        }
    }
}
```

Výpis 7.8: Mapování indexu v databázi Elasticsearch

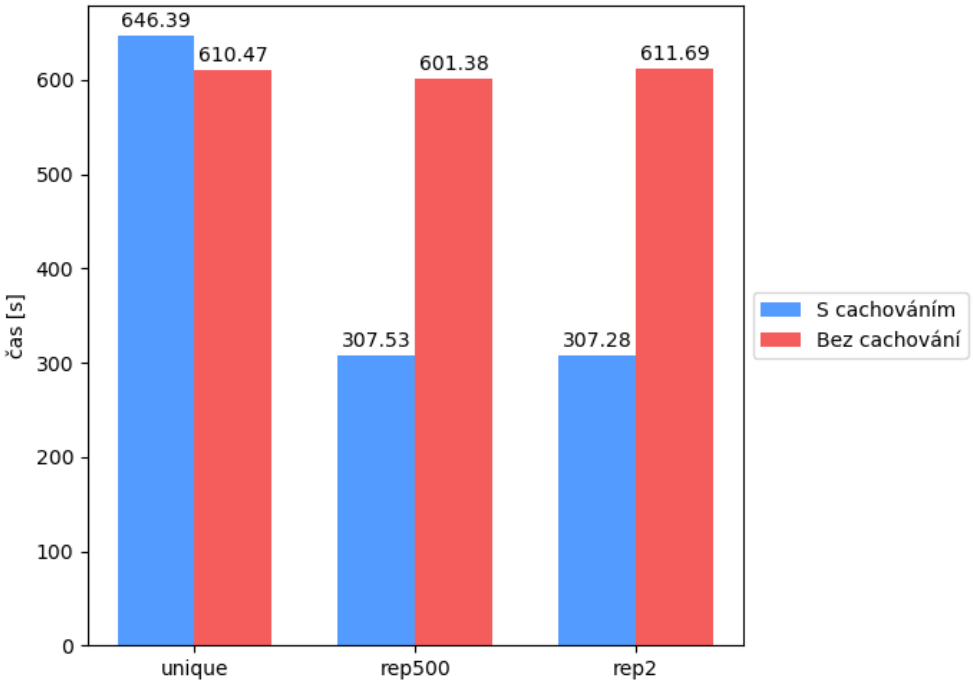
Následně byla do každého z těchto tří vytvořených indexů importována data z databáze gloffer_cache.



(a) Parametrické dotazy



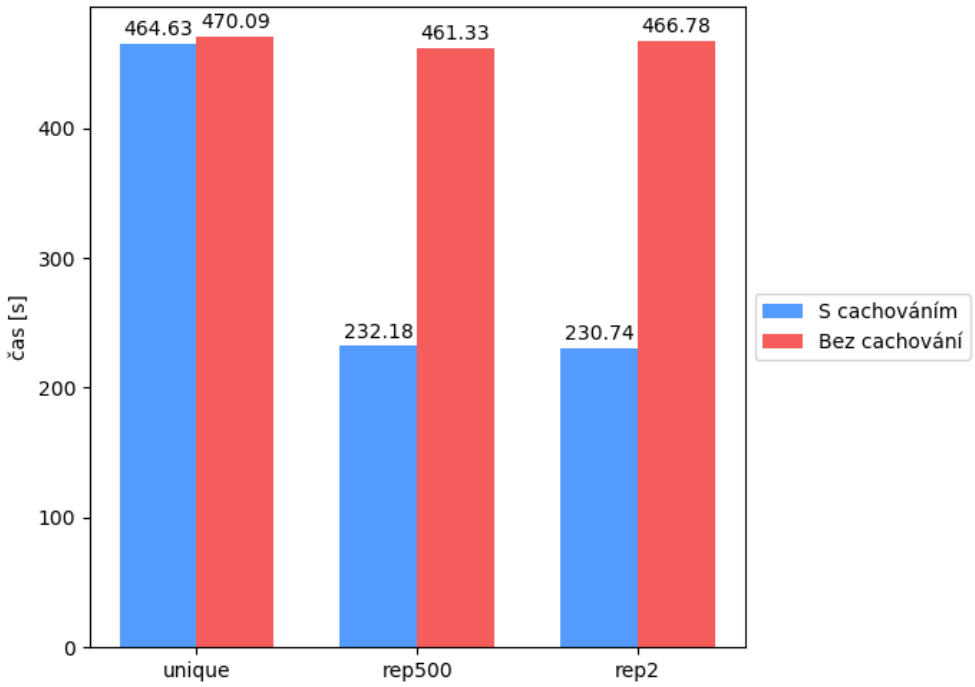
(b) Kombinované dotazy



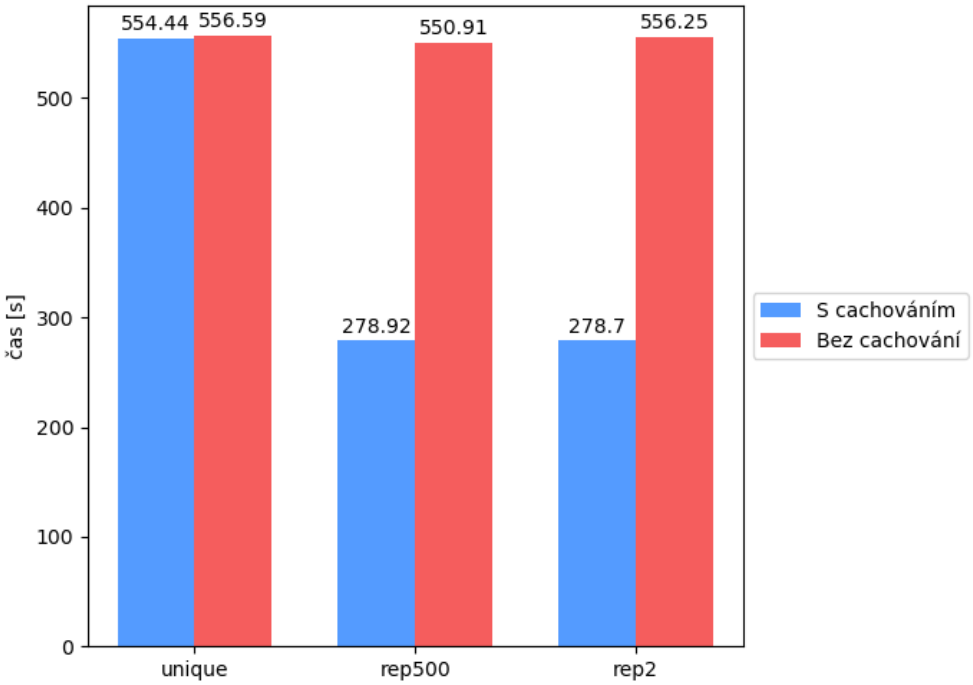
(c) Fulltextové dotazy

Obrázek 7.8: Výsledky měření v databázi Elasticsearch (index obsahující 1 shard)

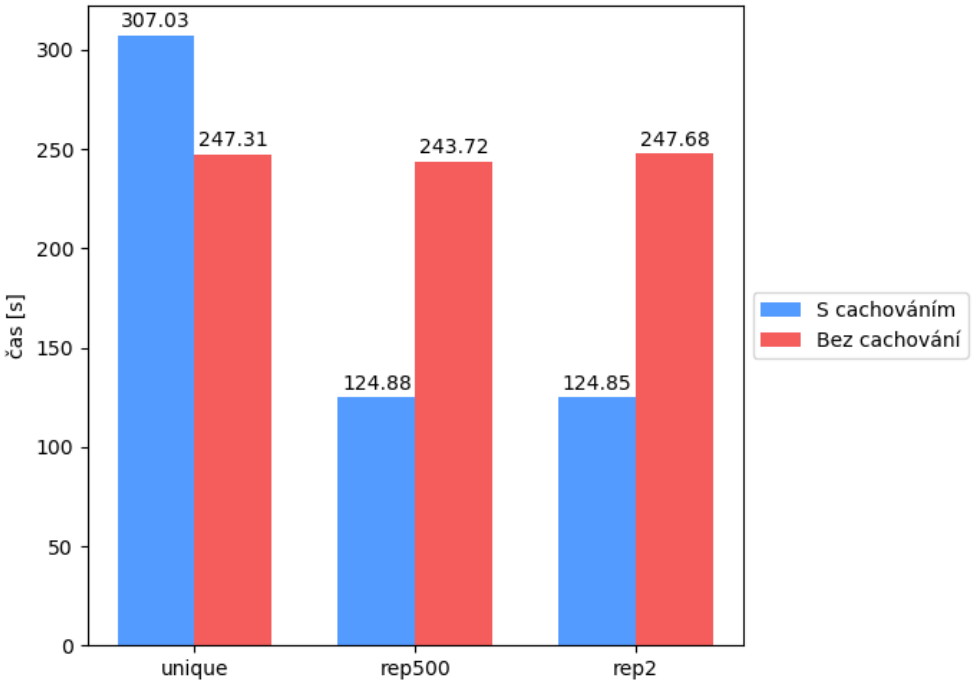
Obrázek 7.8 ukazuje výsledky měření na indexu obsahujícím jediný shard. Lze vidět, že v tomto případě došlo ke zvýšení času vykonávání u spouštění dotazů bez cachování o zhruba 96–106 % u obou skupin opakujících se dotazů, tedy **rep500** i **rep2**, a to u dotazů parametrických, kombinovaných i fulltextových. U kombinovaných dotazů v tomto případě nedošlo ke zvýšení časů oproti pouze parametrického vyhledávání, časy jsou naopak dokonce nepatrně nižší, ovšem rozdíly mezi spouštěním dotazů s cachováním a bez cachování zůstávají podobné. Čistě fulltextové dotazy pak dosahovaly lepších výsledků, což dokazuje, že Elasticsearch je vyladěn zejména pro fulltextové vyhledávání.



(a) Parametrické dotazy



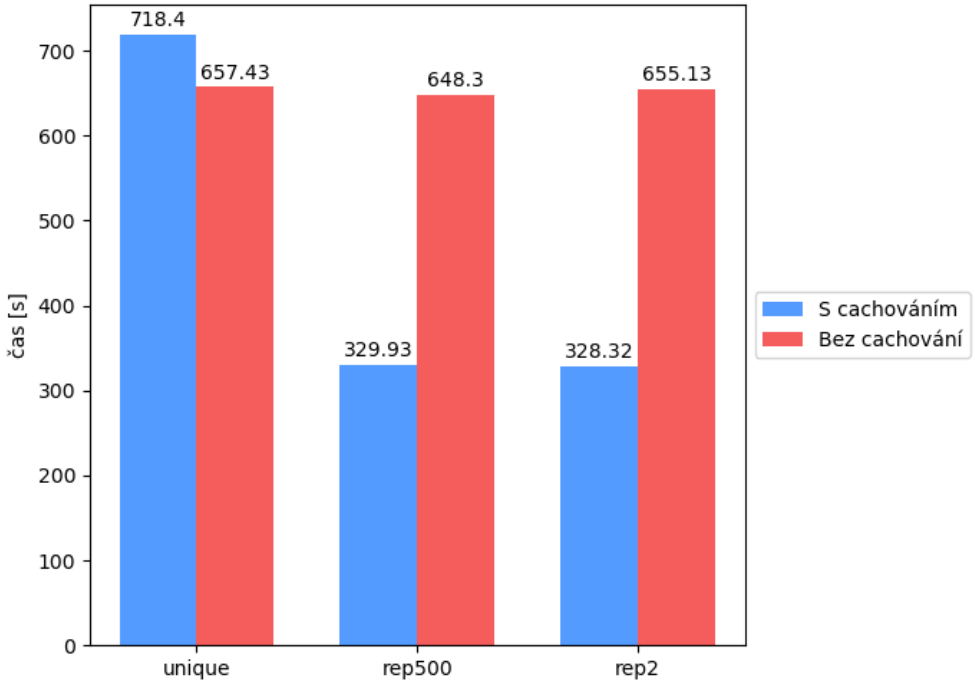
(b) Kombinované dotazy



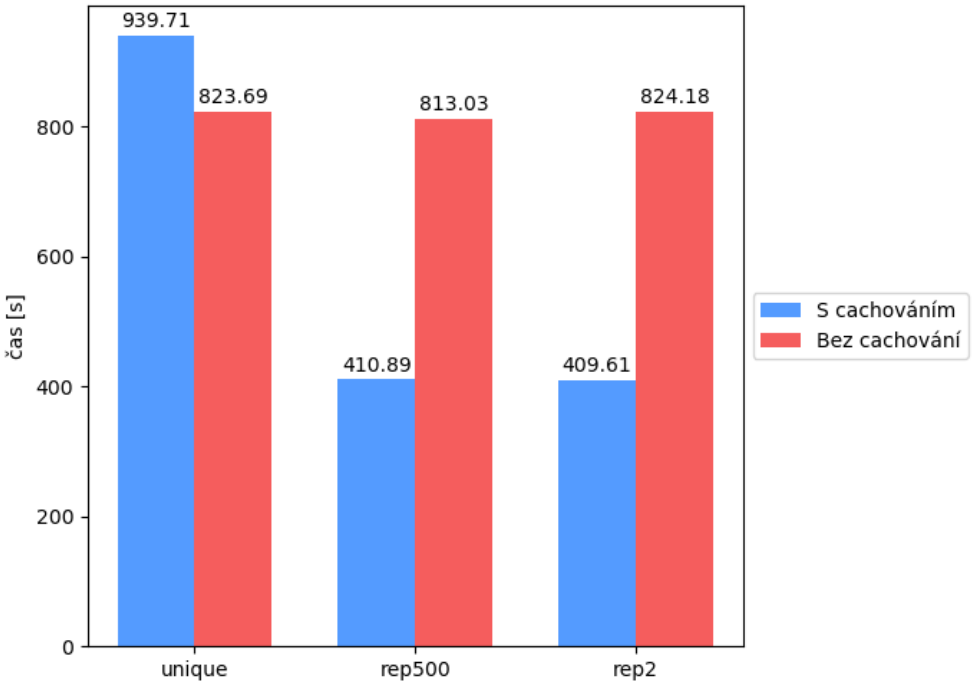
(c) Fulltextové dotazy

Obrázek 7.9: Výsledky měření v databázi Elasticsearch (index s 5 shardy)

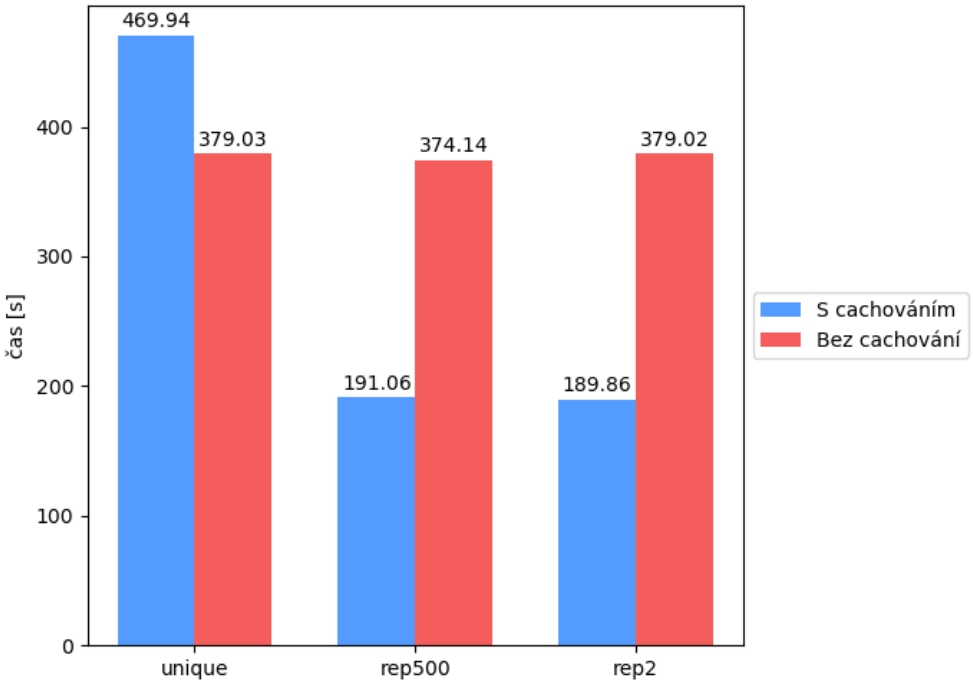
Obrázek 7.9 ukazuje výsledky měření na indexu s 5 shardy a obrázek 7.10 ukazuje výsledky u indexu s 10 shardy. V obou těchto případech došlo ke snížení všech časů vykonávání díky využití paralelizace, lepších výsledků však dosahuje konfigurace s 5 shardy. Ani v jednom případě nedošlo ke změně ohledně rozdílu času vykonávání dotazů s cachováním a bez něj. Stejně jako u předchozího indexu obsahujícího 1 shard, i zde lze vidět zvýšení času bez cachování o 95–102 % u skupin **rep500** a **rep2**, a to opět u dotazů parametrických, kombinovaných i fulltextových



(a) Parametrické dotazy



(b) Kombinované dotazy



(c) Fulltextové dotazy

Obrázek 7.10: Výsledky měření v databázi Elasticsearch (index s 10 shardy)

7.5 Zhodnocení výsledků

Testování ukázalo, že z hlediska parametrických a kombinovaných dotazů vykazují nejlepší výkon databáze MySQL a MongoDB, které tyto dotazy dokázaly vyhodnotit ve srovnatelně krátkém čase. Oproti tomu Elasticsearch dosahoval v tomto ohledu až stonásobně horších výsledků.

Z hlediska fulltextového vyhledávání dopadlo zdaleka nejhůře MongoDB - časy vykonávání těchto dotazů byly velmi výrazně vyšší oproti ostatním databázím. MySQL dokázalo vyhodnotit fulltextové dotazy zhruba 20× pomaleji než parametrické, ale stále za relativně krátkou dobu. Elasticsearch u fulltextových dotazů dosahoval o poznání lepšího výkonu než u dotazů s parametrickým vyhledáváním. Důležité je však připomenout, že Elasticsearch nabízí také velmi široké spektrum možností pro analýzu textu, jeho indexování a vyhledávání v něm. Nepatrně horší výkon oproti MySQL tak vyrovnává fakt, že dokáže vracet mnohem přesnější a relevantnější výsledky.

Ohledně cachování v úložišti Redis ukazují výsledky, že pokud se dotazy často opakují a databázový cache buffer dokáže uchovávat příslušné stránky ve své paměti, tak Redis nepřináší žádné zlepšení, což ukazuje skupina dotazů **rep500**. Avšak pokud jsou dotazy rozmanité, cache buffer musí pracovat s mnoha různými stránkami a dříve použité stránky musí ze své paměti odstraňovat, aby bylo uvolněno místo pro nové stránky. Při položení dotazu podruhé tak nemusí být již příslušné stránky uložené v bufferu a musí být načteny z disku. Tento problém je eliminován cachováním v Redisu, kde tyto klíče zůstávají trvale až do jejich účelného smazání (případně vypršení jejich platnosti, je-li nastavena expirace), viz skupina dotazů **rep2**. Výjimku však tvoří Elasticsearch, jehož časy se bez cachování v Redisu vůbec neměnily - zůstávaly neměnné jak u unikátních dotazů, tak u obou skupin opakujících se dotazů.

Samotný mechanismus cachování v Redisu (vypočítání hash hodnoty, prohledávání cache a ukládání hodnot do cache) nepřinášel žádné nebo jen velmi nepatrné snížení výkonu. To ukazuje, že Redis skutečně dokáže vést ke zvýšení výkonu aplikace v případě, že zde dochází k opakování velkého množství dotazů. Výhodou je také možnost využití distribuované cache, kterou lze lehce nastavit a využít tak například přebytečnou paměť na různých serverech pro cachování. Není tak potřeba zvyšovat paměť pro cache buffer pouze na databázovém serveru. Zároveň použití Redisu přináší větší kontrolu nad cachovanými daty a jejich správou. Je však potřeba mít na paměti, že je při jeho používání potřeba také přinášet mechanismy pro znehodnocování zastaralých dat.

Závěrem lze shrnout, že pro všeobecné využití lze doporučit databázi MySQL, pokud je možné data uchovávat v relační databázi. MySQL dosahuje vynikajícího výkonu pro parametrické vyhledávání a uspokojivého výkonu pro fulltextovou funkcionalitu, avšak tato funkcionalita není v MySQL primární a nemá tak rozšířené možnosti analýzy textu. Pokud je potřeba ukládat data v podobě beze schémových dokumentů, hodí se použití databáze MongoDB, avšak nikoli pro potřeby fulltextového vyhledávání. V případě, že je v dané aplikaci klíčové fulltextové vyhledávání, je ideální použít Elasticsearch, jehož o něco pomalejší vyhledávání vyvažuje schopnost pokročilé manipulace s textovými dokumenty a možnost přizpůsobení analýzy textu potřebám konkrétní aplikace.

Kapitola 8

Závěr

V práci bylo popsáno řešení pro implementaci znovu použitelnosti výsledků vyhledávání nad rozsáhlými databázemi. Byl vysvětlen postup při cachování výsledků databázového hledání a shrnuty základní související principy.

V práci však také byly popsány problémy, které může cachování výsledků vyhledávání přinést, a to zejména problém aktualizace dat v databázi a zastarávání dat v cache. Byly navrženy různé přístupy, které tento problém řeší. V testovací aplikaci byly tyto postupy také implementovány pro názornou ukázku. V tomto ohledu neexistuje univerzálně nejlepší přístup; zvolený postup by se měl odvíjet od potřeb vyvíjené aplikace, zejména od toho, zda je v ní kladen důraz na co nejvyšší výkon nebo na konzistenci dat.

Je potřeba zdůraznit, že v případě, že povaha aplikace vyžaduje trvalou konzistenci dat nebo v ní dochází k častým změnám v datech, je potřeba využívat mechanismy, které data v cache zneplatňují. Tyto mechanismy však snižují efektivitu samotného cachování. Implementace cachování by proto měla být provedena až po zvážení, zda bude pro danou aplikaci přínosem.

V praktické části byly analyzovány a otestovány tři různé databázové systémy. S ohledem na výsledky pak bylo doporučeno MySQL pro ukládání dat v relační podobě, protože dosahuje dobrých výsledků jak pro parametrické tak pro fulltextové dotazy. Technologie Elasticsearch byla doporučena pro systémy založené na fulltextovém vyhledávání, jelikož podporuje široké možnosti analýzy textu a zaměřuje se na vysokou relevanci výsledků vyhledávání. Zároveň byly uvedeny také krokové návody pro instalaci těchto systémů.

Testování dále ukázalo, že ukládání výsledků vyhledávání v cache může výrazně zvýšit výkon aplikace díky deduplikaci vstupních dotazů. Bylo implementováno řešení za použití populární technologie Redis, která vykázala dobrý výkon a byla rovněž doporučena pro praktické použití, jelikož umožňuje opakovaně vracet výsledky předchozích vyhledávání v konstantním čase. V práci byly uvedeny a porovnány také alternativní technologie, které lze k tomuto účelu použít, a to včetně cloudových platforem.

V případě dalšího zkoumání této problematiky je možné se zaměřit na porovnání dalších relačních i dokumentově orientovaných databázových systémů. Je možné rovněž otestovat, zda by došlo u databáze MongoDB ke zlepšení výkonu fulltextového vyhledávání v případě, že by byla indexována data v některém z podporovaných jazyků, například v angličtině. Zajímavé výsledky by mohlo přinést také zasílání dotazů na testovací databáze za použití více vláken, čímž by se otestoval výkon těchto databázových systémů z hlediska paralelizace.

Literatura

1. *DB Engines*. 2021-03-16. Dostupné také z: https://db-engines.com/en/.
2. *Chapter 15: The InnoDB Storage Engine*. 2021-03-07. Dostupné také z: https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html.
3. *Chapter 16: Alternative storage engines*. 2021-03-07. Dostupné také z: https://dev.mysql.com/doc/refman/8.0/en/storage-engines.html.
4. *MARIADB PLATFORM X5 VS. MYSQL ENTERPRISE EDITION 8*. 2021-03-16. Dostupné také z: https://go.mariadb.com/GLBL-WC2020MariaDBvs.MySQL%5C_LP-Registration.html.
5. *The most popular database for modern apps.* 2021-02-10. Dostupné také z: https://www.mongodb.com/.
6. *WiredTiger Storage Engine.* 2021-03-04. Dostupné také z: https://docs.mongodb.com/manual/core/wiredtiger/.
7. *Free and OPEN search: The creators of Elasticsearch, Elk & Kibana.* 2021-02-09. Dostupné také z: https://www.elastic.co/.
8. *Basic Concepts*. 2021-03-26. Dostupné také z: https://www.elastic.co/guide/en/elasticsearch/reference/6.2/_basic_concepts.html#_basic_concepts.
9. KUBANDA, Eduard. *Nástroje pro hromadné generování a předzpracování dat pro fulltextové vyhledávání včetně API*. Ostrava, 2018.
10. *Hunspell: About.* 2021-03-03. Dostupné také z: https://hunspell.github.io/.
11. *Text Analysis.* 2021-03-03. Dostupné také z: https://www.elastic.co/guide/en/elasticsearch/reference/7.11/analysis.html.
12. SHIVANG. *Distributed Cache 101 – The Only Guide You'll Ever Need.* 2020-03. Dostupné také z: https://www.8bitmen.com/distributed-cache-101-the-only-guide-youll-ever-need/.
13. *documentation*. 2021-02-11. Dostupné také z: https://redis.io/documentation.

14. KRÁTKÝ, M.; BAČA, R. *Databázové systémy*. Ostrava, Česká republika: Katedra informatiky, Fakulta elektrotechniky a informatiky VŠB – Technická univerzita Ostrava, 2020.

15. XIE, Y.; O'HALLARON, D. Locality in search engine queries and its implications for caching. *IEEE INFOCOM 2002, 1238-1247*. 2002.

16. MARKATOS, E. P. On caching search engine query results. *Computing Communications 24 (2001), 137-143*. 2001.

17. CAMBAZOGLU, Berkant Barla; JUNQUEIRA, Flavio P.; PLACHOURAS, Vassilis; BANACHOWSKI, Scott; CUI, Baoqiu; LIM, Swee; BRIDGE, Bill. A Refreshing Perspective of Search Engine Caching. In: *Proceedings of the 19th International Conference on World Wide Web*. Raleigh, North Carolina, USA: Association for Computing Machinery, 2010, s. 181–190. WWW '10. ISBN 9781605587998. Dostupné z DOI: 10.1145/1772690.1772710.

18. *Open source database (rdbms) for the enterprise*. 2021-02-10. Dostupné také z: https://mariadb.com/.

19. LABIB, Michael. *Database Caching Strategies Using Redis*. 2017. Dostupné také z: https://d0.awsstatic.com/whitepapers/Database/database-caching-strategies-using-redis.pdf.

20. LORD, Marr. *MySQL 8.0: Retiring Support for the Query Cache.* 2020-04-08. Dostupné také z: https://mysqlserverteam.com/mysql-8-0-retiring-support-for-the-query-cache/.

21. *Redis Cluster Tutorial*. 2021-03-20. Dostupné také z: https://redis.io/topics/cluster-tutorial.

22. BAEZA-YATES, Ricardo; JUNQUEIRA, Flavio; PLACHOURAS, Vassilis; WITSCHEL, Hans Friedrich. Admission Policies for Caches of Search Engine Results. In: 2007-09, sv. 4726, s. 74–85. ISBN 978-3-540-75529-6. Dostupné z DOI: 10.1007/978-3-540-75530-2_7.

23. NATH, K. *All things caching- use cases, benefits, strategies, choosing a caching technology, exploring some popular products*. 2018-12. Dostupné také z: https://medium.com/datadriveninvestor/all-things-caching-use-cases-benefits-strategies-choosing-a-caching-technology-exploring-fa6c1f2e93aa.

24. SIVASUBRAMANIAN, Swaminathan; PIERRE, Guillaume; STEEN, Maarten van; ALONSO, Gustavo. Analysis of Caching and Replication Strategies for Web Applications. *Internet Computing, IEEE*. 2007-02, roč. 11, s. 60–66. Dostupné z DOI: 10.1109/MIC.2007.3.

25. OLSTON, Christopher; MANJHI, Amit; GARROD, Charles; AILAMAKI, Anastasia; MAGGS, Bruce; MOWRY, Todd. A Scalability Service for Dynamic Web Applications. In: 2005-01, s. 56–69.

26. SAZOGLU, Fethi Burak; CAMBAZOGLU, B. Barla; OZCAN, Rifat; ALTINGOVDE, Ismail Sengor; ULUSOY, Özgür. Strategies for Setting Time-to-Live Values in Result Caches. In: *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*. San Francisco, California, USA: Association for Computing Machinery, 2013, s. 1881–1884. CIKM '13. ISBN 9781450322638. Dostupné z DOI: 10.1145/2505515.2507886.

27. *How to discover hotkeys in Redis 4.0.* 2020-04-07. Dostupné také z: https://www.alibabacloud.com/help/doc-detail/101108.htm.

28. DAVISON, A. *Memcached Cache Invalidation Made Easy.* 2009-12. Dostupné také z: http://instantbadger.blogspot.com/2009/12/memcached-cache-invalidation-made-easy.html.

29. *Using Redis as an LRU cache.* 2021-03-28. Dostupné také z: https://redis.io/topics/lru-cache.

30. *Redis Persistence.* 2021-03-28. Dostupné také z: https://redis.io/topics/persistence.

31. *Redis Enterprise.* 2020-04-11. Dostupné také z: https://redislabs.com/redis-enterprise/.

32. DORMANDO. *A distributed memory object caching system.* 2021-02-26. Dostupné také z: https://memcached.org/.

33. *add data replication feature to memcached 1.2.x.* 2021-04-09. Dostupné také z: http://repcached.lab.klab.org/.

34. *JAVA'S most Widely-used CACHE.* 2021-02-26. Dostupné také z: https://www.ehcache.org/.

35. *Amazon ElastiCache for Memcached - Amazon Web Services (AWS).* 2020-04-03. Dostupné také z: https://aws.amazon.com/elasticache/memcached/.

36. *What is Azure Cache for Redis?* 2020-04-09. Dostupné také z: https://docs.microsoft.com/en-us/azure/azure-cache-for-redis/cache-overview.

37. *Redis Quick Start.* 2021-03-25. Dostupné také z: https://redis.io/topics/quickstart.

38. *Text search languages.* 2021-02-16. Dostupné také z: https://docs.mongodb.com/manual/reference/text-search-languages/#text-search-languages.

39. *Setting JVM OPTIONS: Elasticsearch REFERENCE [MASTER].* 2021-03-13. Dostupné také z: https://www.elastic.co/guide/en/elasticsearch/reference/master//jvm-options.html.

40. *Advanced configuration SETTINGS: Elasticsearch REFERENCE [MASTER].* 2021-03-07. Dostupné také z: https://www.elastic.co/guide/en/elasticsearch/reference/master//advanced-configuration.html.